Quantification rythmique dans OpenMusic

Adrien Ycart

Quantification rythmique dans OpenMusic

Auteur :
Adrien Ycart

Référents :
Jean Bresson
Florent Jacquemard

Stage de master ATIAM
réalisé à l'Ircam
du 16 février au 14 août 2015

31 juillet 2015

Résumé

La quantification rythmique est l'action de transformer un flux temporel, par exemple une séquence de notes, en une partition, connaissant le tempo. Cette séquence de notes peut provenir de la captation du jeu d'un musicien ou être générée par un processus compositionnel : c'est ce deuxième cas qui nous intéressera. L'étape de transcription faisant partie du processus créatif, nous cherchons à développer un outil permettant aux compositeurs de transcrire comme ils le souhaitent leurs oeuvres en partitions. Pour cela, nous avons interrogé des compositeurs et nous avons développé un algorithme proposant un ensemble de transcriptions possibles d'une séquence de notes, classées selon un critère prenant en compte la précision de la notation (à quel point la sortie est fidèle à l'entrée) et sa complexité (à quel point la notation est lisible). Nous présentons des résultats de cet algorithme et les comparons à ceux de l'outil actuel de quantification de l'environnement de composition assistée par ordinateur OpenMusic. Nous proposons différentes pistes d'amélioration pour des travaux futurs.

Abstract

Rhythm quantization is the act of transforming a temporal stream, for example a sequence of notes, into a music sheet, given the tempo. This sequence of notes can be taken from the performance of a musician, or generated with computer-assisted composition tools : we will consider the latter case in this work. Transcription is a part of the creation process, thus we aim to develop a tool allowing the composers to transcribe their works into music sheets the way they want. To do so, we interviewed composers and we developed an algorithm that returns a set of possible transcriptions of a sequence of notes, ranked according to a criterion that takes into account the precision of the notation (how close the output is from the input) and its complexity (how readable it is). We present some results of this algorithm and compare them to those returned by the quantization tool currently used in the computer-assisted composition environment OpenMusic. We suggest various possible improvements for future work.

Remerciements

Je remercie mes deux encadrants, Jean Bresson et Florent Jacquemard pour leur aide précieuse tout au long de ce stage, depuis la définition des thématiques de recherche jusqu'à la rédaction de ce rapport. Que ce soit par leur aide dans la compréhension des concepts manipulés, dans l'implémentation des algorithmes développés, ou dans la restitution des travaux effectués, leur recul et leur soutien m'ont été indispensables pour mener à bien ce travail de recherche, et je leur en suis extrêmement reconnaissant.

Je remercie tous les compositeurs et utilisateurs d'OpenMusic que j'ai interrogés pendant ce stage pour leur disponibilité et leurs idées. Leur contact m'a énormément appris à la fois sur leurs méthodes de travail, sur leurs visions respectives de la composition et sur leurs attentes, qui ont permi d'orienter ce stage et de guider mes recherches.

Je remercie Karim Haddad, pour le temps qu'il a eu la gentillesse de consacrer à m'expliquer sa vision du rythme et des outils de quantification, qui m'aura permi de bien cerner le problème.

Enfin, je remercie mes camarades et collègues pour leurs discussions toujours stimulantes, ainsi que l'ensemble du personnel de l'Ircam pour leur accueil chaleureux.

Table des matières

Chapitre 1

La quantification rythmique

La quantification est l'action de convertir des données à valeurs continues (ou discrètes appartenant à un grand ensemble de valeurs) en données à valeurs discrètes appartenant à un petit ensemble de valeurs. Ce problème se pose en particulier lorsque l'on souhaite numériser des données analogiques, telles que par exemple un signal acoustique en un fichier audio, ou un signal lumineux en une image numérique, ou encore lorsque l'on souhaite compresser des données. Cela se fait le plus souvent au prix d'approximations, que l'on souhaite les moins sensibles possible. Dans le domaine de la musique, on se confronte au problème de la quantification lors de la transcription d'une performance, d'un enregistrement, ou de données musicales non quantifiées, en partition. On doit d'une part quantifier les hauteurs, c'est à dire obtenir à partir de la donnée physique de la fréquence fondamentale la notes de la gamme à laquelle elle correspond, d'autre part, on doit quantifier les durées, c'est à dire transformer la suite des instants de début et de fin des notes en rythmes écrits en notation occidentale. Ce stage porte sur ce deuxième problème.

Dans cette section, nous poserons le problème de la quantification, puis nous décrirons quelques formalismes d'arbres de rythme, et enfin, nous ferons un état de l'art des méthodes de transcription rythmique existantes.

1.1 Description du problème

1.1.1 Explication préalable sur le rythme

Avant de décrire les principales difficultés du problème de la quantification rythmique ainsi que l'angle sous lequel nous l'aborderons, il convient d'expliquer ce qu'est le rythme et comment il est défini dans le système de notation occidental.

Le rythme caractérise tout ce qui touche à l'organisation temporelle de la musique. Il correspond au flux temporel des durées des notes et accords d'une pièce de musique.

Dans la musique occidentale, le rythme est défini par rapport à la pulsation. La pulsation, aussi appelée tactus, est la périodicité ressentie lorsque l'on écoute de la musique, celle naturellement battue par un auditeur à qui l'on demanderait de taper des mains en rythme [20]. Le tempo définit la durée de cette pulsation : il donne le nombre d'unités rythmiques de base, en général, la noire, qu'il y a dans une minute. Un tempo de 60 à la noire correspond donc à 60 noires par minutes, c'est-à-dire qu'une noire dure une seconde.

Une fois le tempo fixé, les différentes durées sont définies par division successives de l'unité de temps, comme indiqué Figure 1.1 : une ronde se divise en deux blanches, une blanche se divise en deux noires, une noire se divise en deux croches, une croche se divise en deux doubles croches, et ainsi de suite. Le nombre de divisions d'une unité de temps peut ne pas être égal à deux, par exemple, dans le cas d'un rythme binaire, une noire est divisée en deux croches, mais dans le

cas d'un rythme ternaire, elle est divisée en trois. D'autres divisions moins courantes, appelées irrationnelles, existent également. Le rythme est ensuite donné par l'assemblage de ces figures de notes. Les rythmes sont ensuite regroupés en mesures, dont la durée est donnée par la signature rythmique, afin d'être manipulés plus facilement en unités cohérentes.

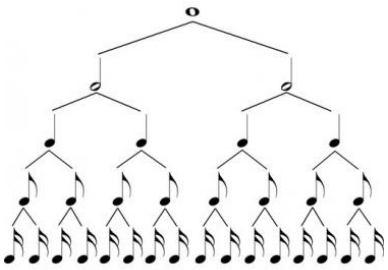

FIGURE 1.1 – Division successive des durées : une ronde vaut deux blanches, une blanche vaut deux noires...

1.1.2 Le problème de la quantification rythmique

De l'explication précédente, on peut déduire qu'il y a une triple dépendance entre les durées, exprimées en secondes, le tempo, exprimé en unité par minute, et les figures de notes, qui appartiennent à un petit ensemble de fractions de l'unité. Déterminer un de ces éléments à partir des deux autres définit trois problèmes : l'interprétation, l'inférence de tempo, et la quantification rythmique, comme indiqué en Figure 1.2. Une difficulté du problème de la transcription rythmique réside dans le fait que l'on ne connait que les durées, et on souhaite obtenir à la fois le tempo et les figures de notes. Or, ces trois paramètres sont très fortement interdépendants : pour en estimer un, on doit connaître les deux autres. Par exemple, on a besoin du tempo pour déterminer la notation rythmique, mais l'estimation du tempo se repose sur une certaine exactitude des instants de début et de fin des notes. En particulier, un tempo ou une signature mal adaptés peuvent donner des transcriptions inutilement compliqués, comme on peut le voir Figure 1.3.

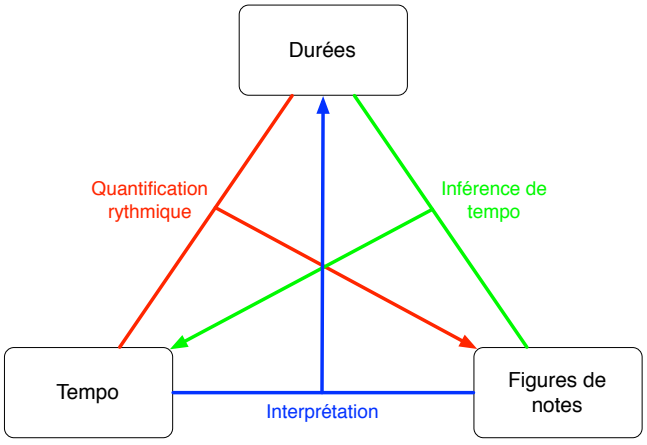

FIGURE 1.2 – Une représentation de la triple dépendance entre durées, tempo et figures de notes, et des trois problèmes qu'elle définit.

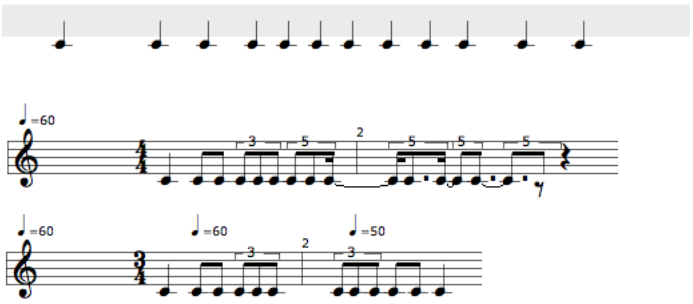

FIGURE 1.3 – On quantifie deux fois la suite de notes en haut, au milieu sans préciser de tempo ni de signature, en bas en précisant les changements de tempo et la signature. On obtient dans le deuxième cas une notation beaucoup plus intelligible.

Dans ce stage, nous nous intéresserons uniquement au problème de la quantification rythmique. Nous supposerons donc que le tempo est connu, qu'il soit donné par l'utilisateur ou détecté à l'aide de méthodes algorithmiques.

Le problème de la quantification rythmique est donc celui de l'obtention, à partir de durées à valeurs réelles et d'une valeur de tempo, de durées exprimées comme des divisions entières de la pulsation : il faudra donc approximer ces durées. Il y a alors un compromis à trouver entre la précision de la notation (c'est-à-dire à quel point la partition est fidèle à l'entrée) et la complexité de la notation (à quel point elle est lisible). En effet, il ne faut pas que la partition dénature ce qui est censé être joué, mais il faut aussi qu'elle soit facilement lisible par un musicien et produise un sens musical. Un exemple est montré Figure 1.4.

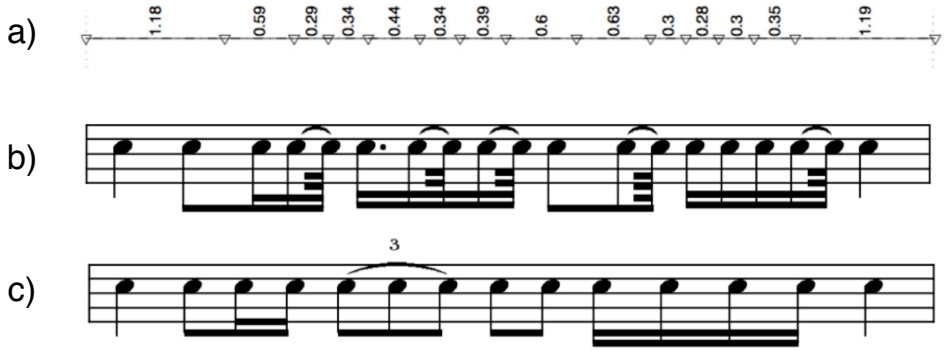

FIGURE 1.4 – Figure extraite de [8] : a) Séquence à quantifier. b) Une notation précise, mais complexe de la séquence. c) Une autre notation de la même séquence, moins précise mais moins complexe.

1.1.3 Deux approches pour la quantification

On peut distinguer deux approches du problème :

Approche "MIR"[1] Les données temporelles sont extraites d'un enregistrement audio d'une performance par des méthodes de traitement du signal, ou d'une captation directe du jeu

1. MIR = Music Information Retrieval

d'un musicien (enregistrement MIDI), et on veut en retrouver la notation rythmique.

Approche "CAO"[2] Les données temporelles ont été générées par un processus compositionnel ou bien dictées par le compositeur sur un clavier, et on veut en faire une partition destinée à être jouée par des interprètes.

La posture à adopter est légèrement différente dans les deux cas. En effet, dans le premier cas, on peut supposer qu'il existe une partition de référence, à laquelle comparer le résultat de la quantification. Dans le deuxième cas, il n'y a pas une seule "bonne" partition que l'algorithme doit donner. En effet, de même qu'une idée peut être écrite de différentes façons, un même rythme peut avoir de nombreuses notations différentes, chacune induisant un sens particulier, une interprétation particulière. Choisir celle qui correspond à ce que le compositeur souhaite transmettre est une étape importante du processus de composition, puisqu'elle conditionne la façon dont les interprètes joueront l'oeuvre et donc le rendu final. L'objectif ici n'est donc pas de trouver la meilleure notation, mais plutôt de fournir au compositeur des outils pour lui permettre d'obtenir la partition qu'il souhaite, parmi un ensemble de partitions possibles choisies selon des paramètres porteurs de sens musical.

1.2 Représentation du rythme

Le rythme peut être considéré comme une succession de durées, une série temporelle. De nombreuses méthodes de transcription rythmique s'en tiennent à cette vision linéaire du rythme. Nous explorerons d'autres formalismes pour représenter le rythme, en particulier les arbres de rythme.

La définition même du rythme repose sur des divisions successives des durées. Cette structure hiérarchique évoque donc naturellement les structures arborescentes (cf Figure 1.1). De fait, la représentation arborescente des rythmes est une idée bien établie, utilisée depuis longtemps [16] et dans différents domaines. Différents formalismes ont été proposés, chacun pour une utilisation particulière. L'une des représentations les plus utilisées est le formalisme d'arbre de rythme d'abord utilisé dans l'environnement de composition assistée par ordinateur PATCHWORK [15], puis dans OpenMusic [3], où les arbres de rythme sont utilisés principalement pour la représentation graphique [2]. Parmi les autres exemples, on peut citer les arbres de rythme symboliques [13] utilisés pour la réécriture de termes, et des représentations qui incluent des éléments mélodiques et rythmiques [22] pour la comparaison d'oeuvres musicales à partir de données symboliques.

Dans ce stage, nous nous concentrerons particulièrement sur les arbres de rythme d'OpenMusic et les arbres symboliques.

1.2.1 Arbres de rythme d'OpenMusic

Les arbres de rythme d'OpenMusic [3] sont étiquetés par des valeurs numériques. On définit une durée de base, puis chaque arbre est écrit selon le même modèle :

$$(D(Subdivisions))$$

D correspond à la durée du noeud, et $(Subdivisions)$ est une liste, éventuellement vide lorsque le noeud est une feuille, contenant des sous-arbres. La durée D d'un noeud est définie en proportions par rapport aux durées de ses cousins. Ainsi, l'arbre de rythme (1 (1 2 1)) est équivalent à (1 (2 4 2)), et donne le rythme ♪♩♪si la durée de base choisie vaut une blanche. Lorsqu'une durée est négative, elle correspond à un silence, et lorsqu'il s'agit d'un flottant et non d'un entier, sa valeur est liée à la précédente.

2. CAO = Composition Assistée par Ordinateur

1.2.2 Arbres de rythme symboliques

Les arbres de rythme symboliques [13], contrairement aux arbres d'OpenMusic, ne contiennent aucune valeur numérique. Les noeuds internes n'ont pas d'étiquettes (ou éventuellement une étiquette correspondant à l'arité du noeud, et donc redondante avec la structure de l'arbre). Les feuilles sont étiquetées avec 4 symboles. Une feuille étiquetée avec le symbole n correspondra à une note, le symbole r donne un silence, le symbole s lie la note avec la précédente, et le symbole 1+ permet de sommer des durées. La durée de chaque feuille est définies par sa position dans l'arbre : la durée d'un noeud est égale à celle de son parent divisée par l'arité du parent, à laquelle on ajoute la durée du cousin de gauche si ce dernier est étiqueté avec le symbole 1+.

La figure 1.5 montre la notation d'un même rythme avec chacun des deux formalismes.

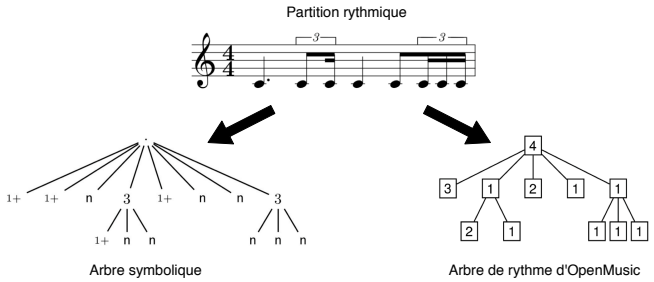

FIGURE 1.5 – Un rythme et ses équivalents dans les deux formalismes d'arbres de rythme décrits

1.3 Etat de l'art

Le problème de la quantification est ancien et complexe. La multiplicité des notations possibles pour une même série de durées en fait un problème d'intelligence artificielle difficile. Dans cette section, nous décrirons les outils qui ont été développés par le passé pour la transcription rythmique, en nous attardant sur deux outils développés à l'Ircam : l'outil de quantification actuel d'OpenMusic, `omquantify`, et l'outil de quantification par réécriture.

De nombreux quantificateurs existent sur le marché, intégrés dans les éditeurs de partitions ou les séquenceurs, mais dans l'immense majorité des cas, les résultats sont peu satisfaisants lorsque les déviations par rapport au rythme original sont trop grandes, ou lorsque les rythmes joués sont trop complexes. D'autres solutions ont été développées, mais aucune n'utilise la structure arborescente du rythme. Certains systèmes se basent sur les rapports entre les durées successives, qui doivent être un rapport d'entiers les plus petits possibles, en particulier celui décrit dans [19]. Cet approche est intéressante, mais l'algorithme décrit fait des hypothèses assez restrictives : le rapport de deux intervalles consécutifs est approximé par un quotient de deux entiers dont un doit être une puissance de 2. Si cette hypothèse est très souvent vérifiée dans le cas de la musique polyphonique, c'est moins systématique quand on considère une seule voix. Ali Cemgil a proposé un modèle bayésien pour la transcription rythmique [6], dans lequel il utilise un modèle de performance où l'erreur entre l'entrée et le rythme idéal est un bruit gaussien. Le modèle de performance le fait plutôt rentrer dans la catégorie des approches "MIR" précédemment évoquées, mais le fait que le bruit soit gaussien et non lié à des considérations sur l'interprétation d'un musicien rend son utilisation possible dans tous les contexte. Cet approche offre des possibilités intéréssantes, comme par exemple celle de pouvoir préciser les divisions successives du temps que l'on autorise, mais l'algorithme est très long à donner des résultats, et ne donne qu'une seule transcription possible à la fois, il ne propose pas plusieurs possibilités. Dans les deux cas, le système se base sur une vision linéaire du rythme, sans profiter de sa structure hiérarchique.

Le principe de fonctionnement de l'algorithme de quantification actuel d'OpenMusic, `omquantify` [18], est décrit dans [1]. Il se base sur un découpage en sous tâches de la séquence à quantifier. La séquence est d'abord découpée en "archi-mesures", puis en mesures, qui correspondent à des segments sur lesquels le tempo est supposé constant. Le tempo peut être déterminé sur chaque segment comme étant un pgcd approximé des durées à l'intérieur du segment. Chaque temps est divisé en n parties égales, avec n allant du nombre de notes dans le temps à 32, et les instants de début des notes sont recalés sur la subdivision la plus proche. Le rythme choisi est celui obtenu à l'aide de la valeur de n, c'est à dire de la grille de quantification offrant le meilleur compromis entre la précision (caractérisée par la distance entre le rythme original et le rythme quantifié) et la complexité (les subdivisions du temps sont classées de la moins complexe à la plus complexe : 1, 2, 4, 3, 6, 5, 8, 7...). La fonction renvoie une structure rythmique, représentée sous la forme d'un arbre de rythme OpenMusic.

Cet outil fonctionne, il est très utilisé par l'ensemble des utilisateurs d'OpenMusic, mais il a plusieurs défauts. Tout d'abord, la grille de quantification est uniforme sur chaque temps, ce qui signifie qu'on ne peut pas quantifier plus finement certaines parties du temps plus denses en notes que d'autres parties de temps qui en contiennent moins. D'autre part, lorsque des contraintes ne peuvent pas être résolues, c'est à dire lorsque deux notes sont recalées sur le même point de la grille, l'une des deux est éliminée, ce qui n'est évidemment pas souhaitable. Il serait préférable d'adopter la notation de "grace note" (appogiature), c'est à dire considérer l'une des deux comme un ornement de l'autre. Enfin, les silences, ne sont pas gérés par l'algorithme, qui prend en entrée une suite de durées et considère que toutes les notes sont consécutives. Pour pouvoir quantifier des silences, il faut les noter explicitement dans la suite des durées d'entrée. Des entretiens avec les utilisateurs de cet outil ont été réalisés au cours de ce stage, dont un compte rendu est donné au chapitre 2, et ont permi de mettre en évidences certains problèmes rencontrés par les utilisateurs de cet outil.

Une autre approche pour la quantification basée sur les arbres de rythme a été proposée [9] et implémentée dans OpenMusic sous forme d'une bibliothèque appelée OMrewrite, s'appuyant sur la notion de règles de réécriture [13], c'est-à-dire des transformations d'arbres définies par filtrage par motif (*pattern matching*). Ces règles permettent d'obtenir différentes notations équivalentes d'un même rythme (c'est-à-dire de la même suite de durées), par exemple pour laisser le choix à l'utilisateur entre différentes notations. Certaines sont conservatives (les durées ne sont pas modifiées), et d'autres non. A partir d'un ensemble de règles de réécriture, on peut proposer un système de quantification, qui commence par générer un arbre de profondeur maximale correspondant à l'entrée, puis applique ces règles pour le simplifier, comme présenté Figure 1.6.

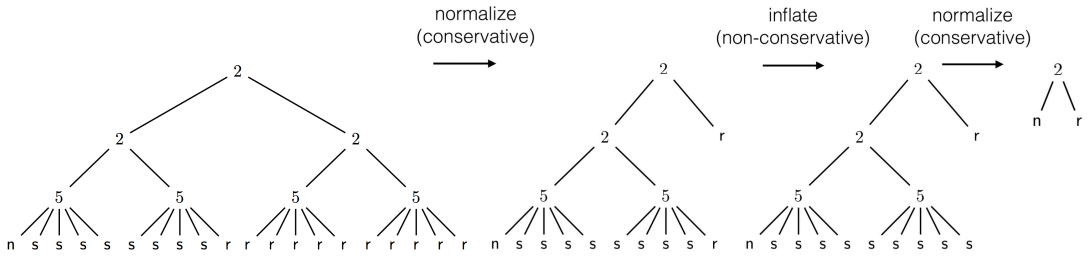

FIGURE 1.6 – Des règles de réécriture sont appliquées à l'arbre complet généré à partir des durées (arbre de gauche). On obtient finalement le rythme "croche demi-soupir".

Ici, la structure arborescente du rythme est bien exploitée, mais le succès de la quantification dépend grandement des règles de réécriture utilisées et de leur application. En effet, le choix de l'ordre dans lequel on applique ces règles n'est pas anodin et peut être déterminant (en d'autres

termes, la propriété de confluence n'est pas forcément vérifiée). Pour avoir un système efficace, il faut choisir une stratégie de réécriture, par exemple de la racine vers les feuilles ou l'inverse. De nombreuses stratégies de réécriture de termes existent [4], la recherche et l'implantation de règles et de stratégies de réécriture pertinentes est un problème de recherche compliqué, qui dépasse le cadre de ce stage. De plus, la structure de l'arbre de profondeur maximale est déterminante pour le résultat final, et elle est figée, on ne peut pas en essayer plusieurs à moins de relancer l'algorithme avec d'autres paramètres.

Chapitre 2

Entretiens avec les utilisateurs d'OpenMusic

Dans le cadre de ce stage, nous avons rencontré quatre utilisateurs de l'environnement OpenMusic. Nous les avons interrogés sur leurs travaux, ainsi que la place qu'y prennent les outils de quantification et de manipulation de rythmes. Nous avons également discuté des problèmes qu'ils ont rencontrés, et des perspectives d'amélioration qui pourraient rendre ces outils plus efficaces à la fois pour les utilisateurs interrogés et, dans un cadre plus général, pour l'ensemble de la communauté OpenMusic. Nous ne nommerons pas les utilisateurs, nous les désignerons simplement par leurs initiales.

2.1 Utilisateur 1 : AR

2.1.1 Description de ses travaux

AR est un compositeur italien qui travaille sur *il canto in ottava rima*, une forme de chant traditionnel italien. Il s'agit de chants improvisés sur des textes poétiques. Les performances s'articulaient sous forme de joutes verbales codifiées, avec des échanges, de appels, auxquels correspondaient certaines formules mélodiques et rythmiques. Il souhaiterait obtenir des partitions de ces chants, qui font partie de la tradition orale italienne et n'ont donc jamais été écrites, afin d'y puiser des idées et enrichir son propre vocabulaire musical.

2.1.2 Méthode de travail actuelle

Son processus de travail est le suivant :

Conversion du fichier audio en MIDI : Pour cela, il utilise les logiciels Audiosculpt et Melodyne. Melodyne donne un fichier MIDI plus lissé, où par exemple les trilles et vibrati sont considérés comme une seule note, tandis qu'AudioSculpt donne un fichier plus proche de l'analyse audio, mais qui peut comporter des artefacts ou considérer un vibrato comme une suite de notes différentes. AR garde les deux approches, car selon lui, même les artefacts obtenus sont intéressants à étudier et constituent un matériau pour son processus de création.

Ajustement du tempo : Le but est d'obtenir un fichier MIDI à tempo constant. Il s'agit d'un étape de pré-quantification, qui est réalisée avec le logiciel Cubase. Cette étape se décompose en plusieurs actions :

— On choisit un premier tempo approximatif à partir des premières mesures du morceau.

— On fixe les durées en secondes des évènements MIDI, et on décale à la souris les pulsations pour les faire coïncider avec des débuts de notes MIDI. Dans cette étape, on ne recale que les notes qui correspondent à des temps forts. Par exemple, pour une longue phrase, on ne recalera que le début et la fin de la phrase, le milieu sera laissé tel quel. On obtient ainsi une piste avec des notes de durées inchangées, mais avec un tempo variable.

— On fixe ensuite les durées des notes non plus par rapport au temps physique [1], mais par rapport à la mesure, puis on rend le tempo constant. Cette étape conserve les coïncidences entre les notes et la pulsation, mais modifie les durées en secondes des notes. On obtient ainsi un fichier MIDI à tempo constant, où les temps forts de la musique sont recalés sur les pulsations.

La Figure 2.2 détaille cette étape.

Quantification dans OpenMusic : AR charge le fichier MIDI ajusté dans OpenMusic, puis il place éventuellement les séparateurs juste avant les temps forts de la musique (cette étape n'est pas forcément obligatoire, puisque ces temps forts sont en général déjà recalés sur des débuts de mesure. Il quantifie ensuite les segments à l'aide d'`omquantify`, en prenant comme tempo celui utilisé lors du réajustement du fichier MIDI. Les autres paramètres sont ajustés à la main, en visualisant le résultat des différentes modifications.

Les différentes étapes de ce processus sont résumées dans le diagramme Figure 2.1

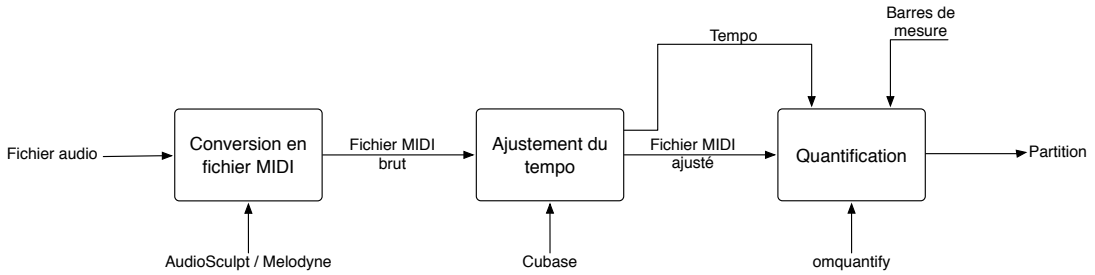

FIGURE 2.1 – Processus de travail d'AR

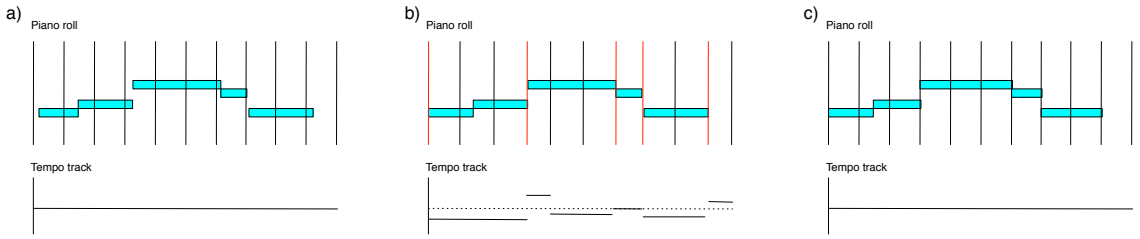

FIGURE 2.2 – Etape d'ajustement du tempo. a) Durées non ajustées, tempo constant. Les barres verticales correspondent aux pulsations. b) Les durées ne sont pas modifiées, le tempo n'est plus constant. Les barres rouges correspondent aux pulsations décalées manuellement. c) On rend le tempo constant. Les durées sont modifiées, les coincidences entre les notes et la pulsation sont conservées.

1. On distingue le temps absolu, physique, qui correspond à des durées en secondes, et le temps relatif, musical, qui est lié à la pulsation et dépend du tempo.

2.1.3 Problématiques

On voit que à différents moments, le compositeur est amené à faire des choix dans le processus de quantification (par exemple pour choisir quels sont les notes à recaler sur des pulsations, quelles notes laisser libres, comment regrouper les notes en mesures). Ces choix ne sont pas anodins, il s'agit de gestes créatifs forts, qui donnent un sens particulier à la musique qui est écrite. Une phrase musicale n'aura pas le même sens, ne sera pas interprétée de la même façon en fonction de la position des temps forts, définis par le compositeur lors de l'écriture de la partition. Même si on pourrait imaginer des algorithmes permettant de répondre à certaines de ces question, par exemple des algorithmes permettant de maximiser les coïncidences entre la grille et les notes, ce n'est pas souhaitable, car c'est au compositeur qu'il revient de prendre ces décisions.

D'autre part, les différents choix faits ont tous une importance sur la partition finale, et ils sont pour la plupart liés entre eux. Le tempo que l'on choisit dans Cubase sera conservé pour tout le reste de la quantification, et le choix de ce tempo se base sur un certain nombre d'hypothèses faites sur la quantification du morceau (en particulier, que les temps forts sont joués sans imprécision). De plus, il n'y a pas une unique solution pour chaque problème, le compositeur peut vouloir essayer plusieurs valeurs de paramètres, et choisir la solution qui lui convient le mieux. On voit qu'avec ce genre de processus de travail, linéaire, on peut difficilement jouer indépendamment sur chaque paramètre : on ne peut pas en faire varier un et obtenir le résultat sans repasser par toutes les étapes du processus.

Enfin, il arrive que l'on veuille transcrire des phrases musicales complexes, de durées différentes, où les temps forts ne sont pas forcément uniformément espacés. Dans ce cas, la transcription peut se faire selon plusieurs paradigmes : soit on préfère faire varier le tempo, soit on fait au contraire varier la signature rythmique. Par exemple, en 4/4, pour noter une mesure constituée de cinq notes de longueur égales, on pourra soit conserver la mesure 4/4, et noter un quintolet, soit utiliser exceptionnellement une mesure à 5/4. Dans le premier cas, il faudra que le tempo soit plus lent que dans le deuxième pour obtenir un résultat équivalent. Ces deux notations ont des sens différents, et peuvent donner lieu à des interprétations différentes (par exemple, le quintolet pourra être interprété plus librement rythmiquement que les cinq noires, où il sera plus important que les 5 notes aient des longueur égales car elles sont directement liées à la pulsation), et un compositeur peut vouloir choisir entre les deux.

2.1.4 Pistes d'amélioration

2.1.4.1 Interface

On se rend compte que dans la méthode de travail d'AR, de nombreux éléments sont ajustés à la main, en fonction du résultats, de la partition qu'ils produisent. Un point important serait de pouvoir facilement modifier chacun des paramètres et des choix qui ont été faits dans une interface unifiée. On éviterait ainsi de nombreux allers retours entre les différents logiciels (en particulier entre Cubase et OpenMusic), et on pourrait voir rapidement l'influence qu'ont les changements effectués sur la partition (par exemple, si on décide d'aligner les notes sur le temps différemment, il est pénible d'avoir à reprendre tout le processus avant de voir comment ces changements se traduisent sur la partition).

Un exemple précis : Pour ajuster le fichier MIDI, AR peut vouloir essayer différents ajustements, en cherchant par exemple à faire coïncider au mieux, entre deux notes fixées sur des temps, les notes et les temps du métronome. Pour faire ça, il est obligé d'effectuer ses modifications dans Cubase, exporter le fichier MIDI, et quantifier dans OpenMusic. Si il pouvait faire directement cette opération dans OpenMusic, il trouverait plus facilement le meilleur ajustement.

2.1.4.2 Modes de quantification

Pouvoir choisir quel paramètre on préfère fixer, et quel paramètre on veut faire varier pourrait être un contrôle important pour la quantification, comme nous l'avons expliqué dans la partie précédente. Pour noter un rythme complexe, avec des mesures de durées différentes, on peut :

- Fixer le tempo et la signature rythmique. On obtient une notation souvent complexe, difficile à déchiffrer, qui ne reflète pas forcément le sens musical de la phrase
- Fixer la signature rythmique, et faire varier le tempo d'une phrase à l'autre. Cette notation peut donner lieu à des rythmes complexes mais a l'avantage de découper en unités cohérentes la musique
- Fixer le tempo, et faire varier la signature rythmique. Si on arrive à trouver un diviseur commun, ou au moins approximé, à toutes les durées de l'ensemble des phrases à quantifier, on peut vouloir utiliser cette unité comme base commune pour la transcription, écrire les rythmes par rapport à cette base et faire varier la signature en fonction de la longueur de chaque phrase. Ce choix va en général donner lieu à des rythmes plus simples à déchiffrer, mais à des mesures aux signatures plutôt complexes et changeantes.

Un exemple est montré Figure 2.3

FIGURE 2.3 – Deux grilles différentes pour quantifier des phrases de durées différentes

2.1.4.3 Silences

Au cours de ces travaux, AR a remarqué que souvent, la durée des silences, des respirations, n'était pas quelconque, elle faisait partie de la musique. Ces respirations ont un sens sémantique, qui n'est pas vraiment analysé dans l'état actuel, car seules les durées des notes sont prises en compte, et on les suppose toutes consécutives. Il n'y a pas donc de quantification des silences à proprement parler si l'on utilise l'outil `omquantify` tel quel. Il existe néanmoins un moyen de quantifier des silences. On peut appliquer au préalable la fonction `true-durations` à l'entrée à quantifier. `true-durations` est une fonction qui prend en entrée un objet `chord-seq`, c'est à dire une séquence de notes ou d'accords, et renvoie une liste de durée. Cette liste de durées est telle que les silences qui étaient présents dans l'objet `chord-seq` sont rendus explicites (ils sont notés avec des durées négatives) et lorsque deux notes consécutives se superposent dans l'objet `chord-seq`, la première est raccourcie jusqu'au début de la deuxième. On peut ainsi quantifier correctement une partition monophonique contenant des silences, puisque les durées des silences sont explicitées, et seront quantifiées comme les autres durées, mais cette fonction est méconnue des utilisateurs.

2.2 Utilisateur 2 : OM

2.2.1 Description de ses travaux

OM étudie dans le cadre de sa thèse en musicologie le chant non mélodique francophone dans le rap, le ragga et le punk. Il essaie en particulier de définir ces styles, trouver des différences et des points communs entre ces courants, et mettre en évidence leurs singularités. Pour cela, il étudie en particulier le placement rythmique de la voix dans ces styles (temps forts, débuts et fins de phrases), et à cette fin, utilise des outils de quantification rythmique.

2.2.2 Méthode de travail actuelle

Son processus de travail est le suivant :

Extraction de syllabes : Pour cela, il utilise AudioSculpt. Il extrait automatiquement les phonèmes et recale à la main leurs instants de début pour les faire correspondre à l'instant de début perçu. Il obtient ainsi une liste d'instants de débuts de phonèmes, ainsi que diverses autres annotations qui n'interviennent pas dans le processus de quantification.

Extraction de beats : Toujours dans AudioSculpt, OM utilise Ircam Beats pour détecter le tempo, ainsi que les mesures et les premiers temps dans ses morceaux. Il obtient ainsi une liste d'instants correspondant aux pulsations, regroupées par quatre (par mesure).

Quantification : Pour quantifier, il récupère les premiers temps de chaque mesure et utilise ces marqueurs pour découper le fichier d'entrée en mesures. Il quantifie ensuite avec `omquantify` sur chaque mesure. Il obtient ainsi une partition par mesure, puis les concatène pour obtenir la partition finale. Les rythmes obtenus peuvent parfois être inutilement compliqués, il interdit donc les subdivisions par 5 et par 7, et obtient des résultats satisfaisants.

Son processus de travail est résumé sur le diagramme Figure 2.4

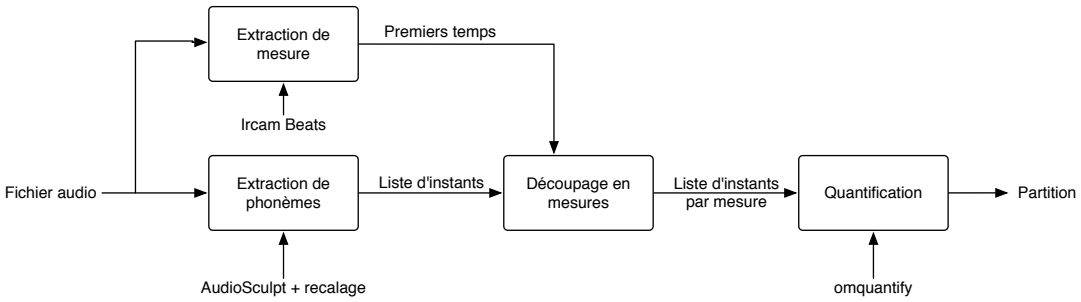

FIGURE 2.4 – Processus de travail d'OM

2.2.3 Pistes d'amélioration

Dans l'état actuel, la quantification est satisfaisante pour OM. Les rythmes obtenus sont à la fois fidèles à ceux qui ont été chantés, et ne donnent pas lieu à des notations inutilement compliquées. Le processus de travail est réalisé par un patch OpenMusic qui fonctionne bien et dont les différents paramètres sont facilement accessibles et modifiables.

Le seul inconvénient est que les séparateurs utilisés pour découper le fichier d'entrée en mesure doivent être forcément liés à une note. Or il n'y a pas forcément une note chantée sur le premier temps de la mesure. Dans ce cas, le séparateur n'est pas pris en compte. Ce cas se produit rarement, donc on n'a jamais plus de deux ou trois mesures regroupées en un segment Kant, cela ne pose donc pas trop de problèmes pour cette application précise. Il pourrait cependant être utile dans d'autres cas de ne pas être contraint de lier un séparateur nécessairement à un segment, mais de pouvoir le placer n'importe où, y compris lorsqu'il ne s'agit pas d'un début de notes (silence, note tenue).

On voit aussi que ici, la quantification est faite indépendamment pour chaque mesure (les paramètres utilisés sont les mêmes pour toutes les mesures, mais l'algorithme de quantification n'a aucune connaissance sur le morceau au delà de la mesure considérée). Ici, les résultats sont satisfaisants, et les rythmes sont assez simples. Cependant, avoir une approche plus globale, en prenant en compte les choix de quantification faits dans les autres mesures pourrait donner de meilleurs résultats dans le cas général.

2.3 Utilisateur 3 : EP

2.3.1 Description de ses travaux

EP est un compositeur italien. Il souhaite composer une oeuvre en prenant comme base la décomposition harmonique d'un son de clarinette.

2.3.2 Méthode de travail

Pour obtenir sa partition, EP commence par analyser le son de clarinette sous AudioSculpt, et en déduit les fréquences de chaque harmonique. Il crée ensuite plusieurs voix, chacune jouant en boucle une note correspondant à un harmonique. La période de répétition de chaque note est une fonction de sa hauteur : plus une note est grave, plus la période sera grande. Il obtient ainsi une partition polyrythmique. Il retravaille ensuite ce matériau en supprimant des groupes de notes pour obtenir la partition qu'il désire.

Cette oeuvre a une structure par définition assez lacunaire, constituée principalement de notes tenues et de silences. Comme par défaut, `omquantify` considère que toutes les notes sont liées les unes aux autres, il utilise ensuite le quantificateur de bach (bibliothèque de CAO pour MaxMSP [24]) pour obtenir une partition de son oeuvre.

2.3.3 Pistes d'amélioration

Le point le plus évident qui ressort de cet entretien est la diffictulté de la gestion des silences. On ne peut pas ici quantifier qu'avec les durées, en considérant que toutes les notes sont liées sans corrompre l'essence même de l'oeuvre. Il faut que les silences soient pris en compte au même titre que les notes, car ils sont, comme ces dernières, des éléments indispensables de la musique. En particulier, lorsque la partition commence par un silence, il ne faudrait pas qu'il soit éliminé, car on risquerait de décaler les voix entre elles, compromettant le bon rendu de la pièce. Ce problème peut être résolu à l'aide de la fonction `true-durations` déjà évoquée. Or cette fonction est souvent inconnue des compositeurs, alors même qu'elle répond exactement à leurs besoins. Il pourrait être utile de la rendre plus facilement visible, ou de l'intégrer mieux aux outils déjà existants.

Un autre point qui a été soulevé est la prise en compte ou non de variations linéaires de tempo (accelerando, rallentando). Pouvoir préciser au quantificateur les sections sur lesquelles on veut que le tempo reste constant, et celles sur lesquelles le tempo peut varier linéairement simplifierait les notations dans le cas où cette variation de tempo est décidée à l'avance (cela évite en particulier de définir un nouveau tempo à chaque mesure avec un faible incrément de pulsations par minutes).

2.4 Utilisateur 4 : GH

2.4.1 Description de ses travaux

GH est un compositeur canadien. Il ne travaille pas directement sur `omquantify`, mais plutôt autour de la représentation du rythme dans OpenMusic. Il travaille sur la génération de phrases musicales à l'aide d'algorithmes génétiques, et cherche en particulier à trouver une bonne représentation pour pouvoir générer des rythmes avec de tels algorithmes.

2.4.2 Algorithmes génétiques

Les algorithmes génétiques visent à modéliser une évolution similaire à a sélection naturelle. Le principe général de tels algorithmes est le suivant :

— On a une population de départ, qui correspond à des solutions possibles d'un certain problème (dans notre cas, par exemple, des séquences de notes). Elle peut être soit initialisée aléatoirement, soit choisie par l'utilisateur.
— On effectue des croisements entre les individus de cette population. Cela revient à créer de nouveaux individus, mélangeant des caractéristiques d'individus existants.
— On applique des mutations sur certains de ces individus. Cela correspond à effectuer de petites modifications sur les individus.
— On sélectionne selon un certain critère les meilleurs individus de la population. On applique à nouveau l'algorithme sur cette population réduite, et on répète cette séquence d'étapes jusqu'à ce que le résultat soit satisfaisant.

A partir de cet algorithme général, il faut répondre à 4 questions pour obtenir un algorithme appliquable à la génération de rythmes :
— Comment effectuer les croisements entre individus ?
— Comment faire les mutations ? La question sous-jacente est celle de la détermination du sens qu'on donne à "petite modification".
— Comment choisir les meilleurs individus ?
— Quelle représentation des rythmes choisir pour effectuer ces modifications ?

2.4.3 Choix de la représentation

Le problème du choix de la représentation est à la fois central et délicat, car il est très lié à la définition des croisements et mutations.

La première option consiste à réutiliser le formalisme des arbres de rythme déjà utilisé dans OpenMusic. Cette représentation est bonne car elle tient compte de la structure intrinsèquement arborescente du rythme. Cependant, un problème de continuité entre la représentation et le résultat musical se pose : une petite modification de la structure de l'arbre peut entraîner une notation, et donc un résultat musical très éloigné (si on touche par exemple à un noeud interne de l'arbre), et de même, une petite modification du résultat musical peut donner un arbre complètement différent (si par exemple on retarde un peu une note jouée sur un temps, il faudra représenter un rythme avec une granularité beaucoup plus fine, et donc beaucoup augmenter la profondeur de l'arbre). Par contre, avec ce formalisme, il est assez facile d'imaginer des croisements entre arbres. On peut par exemple imaginer pour cela qu'on intervertit deux sous-arbres ayant leurs racines au même niveau dans l'arbre, et en prenant tout jusqu'aux feuilles.

La deuxième option consiste à s'affranchir de cette structure arborescente, et de ne considérer que le résultat musical. Ici, il est plus facile de définir une mutation (par exemple, un petit déplacement d'un onset ou d'un offset). Cependant, on perd la structure arborescente, et on risque d'obtenir des résultats peu musicaux. C'est la solution envisagée jusqu'à présent par GH. Dans son formalisme, un rythme est représenté par :
— Un nombre donnant la granularité de la grille
— Un nombre binaire de longueur égale au nombre précédent, où un 1 signifie qu'une note est jouée, un 0 correspond à un silence.

Par exemple, (4 1101) correspond au rythme "noire noire soupir noire". Dans ce cas, les croisements peuvent poser des problèmes. Si par exemple notre séquence est composée de plusieurs couples (granularité ;nombre binaire), on ne pourra pas intervertir n'importe quel couple de sous-séquences car on risquerait de corrompre le format de la séquence.

2.4.4 Utilisation des algorithmes génétiques pour la quantification

Les algorithmes génétiques permettent de résoudre de nombreux problèmes, en fonction des choix de fonctions de sélection, de mutation et de croisement. Si l'on veut les appliquer au problème de la quantification, il faudra prendre des précautions quant à ces fonctions. La fonction de sélection

devra être une fonction évaluant une distance par rapport à la séquence non quantifiée, pondérée par la complexité du rythme quantifié. Les fonctions de mutation et de croisement devront en particulier garantir que le nombre de notes n'est pas modifié. Cela implique que la mutation ne devra pas rajouter ni enlever de notes (c'est à dire de feuilles). On pourra donc par exemple modifier la durée d'un noeud de l'arbre (ce qui change la durée relative de la section descendant de ce noeud), ou encore modifier la structure de l'arbre sans changer le nombre de feuille (par exemple en supprimant un noeud interne et en faisant remonter tous ses fils). La fonction de croisement devra faire en sorte de n'échanger que des motifs (c'est à dire des sous-arbres) contenant le même nombre de notes.

Les algorithmes génétiques peuvent proposer une alternative à `omquantify`, comme nous l'avons vu plus haut, à quelques restrictions près (notamment sur le nombre de notes dans la séquence). Avec des fonctions de mutation et de sélection bien pensées, on pourrait peut-être trouver plus facilement une bonne notation pour une entrée particulière, fidèle, simple, et potentiellement assez différente de celle obtenue avec `omquantify` (la structure de l'arbre dépendra de ses différentes mutations, et non d'un schéma de subdivision prédéfini). Cependant, il n'est pas forcément judicieux d'utiliser ces algorithmes dans ce but. En effet, les algorithmes génétiques sont assez lourds en calcul, et convergent en général au bout d'un temps assez long. Or, nous l'avons vu, les compositeurs ne recherchent pas forcément l'outil qui leur renverra la meilleure partition pour leur oeuvre, ils recherchent plutôt un outil permettant de tester les différents paramètres et visualiser des résultats correspondants facilement et rapidement. Cela peut donc être une alternative, mais cela ne semble pas être la direction préconisée par les compositeurs rencontrés.

2.5 Conclusion

Les principales conclusions que nous pouvons dégager de ces entretiens sont :

Contrôle du compositeur : Le choix d'une notation n'est pas anodin, et fait partie du processus de composition. De plus, les différentes étapes de la production d'une partition ont toutes une grande importance sur le résultat final. Il est donc important que les outils fournis facilitent le contrôle du compositeur sur la partition finale, par exemple en lui permettant de visualiser facilement le résultat des différentes modifications ou en lui proposant de choisir parmi plusieurs solutions possibles au lieu de donner une seule solution.

Gestion des silences et des "grace notes" : Ces aspects de la quantification sont assez mal exploités par les utilisateurs. `omquantify` ne gère pour l'instant pas directement les silences dans les suites de notes, puisque seuls les onsets sont pris en compte. Néanmoins, il existe des fonctions permettant de quantifier les silences, mais elles ne sont en général pas connues des utilisateurs. De même, certaines méthodes existent pour quantifier les "grace notes", mais le formalisme des arbres de rythme ne le prend pas en compte.

Fixer les temps forts : Il n'existe pas de méthode simple permettant de définir quelles notes sont censées tomber sur des temps, où doivent être les barres de mesure, ainsi que d'autres paramètres importants du point de vue sémantique (qu'il revient au compositeur de définir et non à un algorithme).

Tactus, signature, tempo : Lorsque l'on veut quantifier des phrases de durées (en seconde) différentes, on peut soit garder un tempo constant et faire varier la signature rythmique, soit faire varier le tempo en gardant une signature constante. Un schéma représentant ces deux modes de quantification est donné Figure 2.3. On peut aussi préférer noter des variations linéaires du tempo (accelerando, decelerando), plutôt que de faire appel à des rythmes ou des notations plus complexes. Les compositeurs sont demandeurs d'outils permettant de naviguer entre ces différentes notations.

Chapitre 3

Automates d'arbres et quantification

Suite aux entretiens réalisés, nous avons essayé de développer un outil plus performant vis à vis des problèmes évoqués. Notre système prend en compte les silences et les "grace notes". Nous avons également proposé une solution au problème du contrôle du musicien en développant un algorithme permettant au compositeur de choisir entre plusieurs quantifications possibles pour une série de durées donnée, et ce avec peu de calculs. Cette solution a été développée en se basant sur la classe d'algorithmes, appelés algorithmes k-best, décrite dans [12]. Ils permettent, à l'aide de techniques de programmation dynamique, de renvoyer, parmi une classe d'arbres prédéfinie, les arbres de poids les plus faibles selon une certaine mesure. Les poids sont évalués de façon paresseuse, ce qui rend ces algorithmes efficaces d'un point de vue computationnel. De plus, ces algorithmes permettent une structuration du rythme plus complexe que la division de la pulsation en partie de durées égales proposée par `omquantify`.

Dans cette partie, nous commencerons par évoquer les points théoriques sur lesquels notre algorithme se repose, puis nous décrirons notre version de l'algorithme k-best ainsi que les modifications que nous lui avons apportées pour l'adapter à la quantification rythmique.

3.1 Fondements théoriques

Les arbres de rythme dans OpenMusic peuvent être vus comme des arbres de dérivation de grammaires hors-contexte. Le fait de les considérer comme tels ouvre la voie à l'utilisation des automates d'arbres et à tous les outils qui ont été développés autour de ce formalisme [7].

3.1.1 Automates d'arbres et grammaires hors-contexte

Les automates d'arbres sont un type de machines à états finis. Ils sont très similaires aux automates standards, qui opèrent sur les mots [11], à ceci près que les transitions ne se font pas d'un état à un autre état, mais d'un état à un ensemble d'états. De même qu'un automate permet de reconnaitre ou de générer les mots qui appartiennent à un langage régulier, un automate d'arbres permet de reconnaitre ou de générer les arbres qui appartiennent à un langage régulier d'arbres ou, de manière équivalente, reconnaitre l'ensemble des arbres de dérivation des mots générés par une grammaire hors-contexte. Nous utiliserons donc les automates d'arbres pour générer des arbres de dérivation d'une grammaire hors-contexte. Chaque noeud de l'arbre est associé, lors du calcul de l'automate, à un état de l'automate, qui correspond à un symbole de la grammaire hors-contexte. Cette grammaire est définie par un schéma de subdivision, qui décrit les subdivisions successives possibles des unités de temps. Un exemple explicitant le lien entre rythme et grammaires hors-contexte est donné Figure 3.1 : la grammaire hors contexte est donnée à gauche, le mot généré

par la grammaire est la suite des durées donnée par les feuilles de l'arbre, et l'arbre de rythme de sortie est l'arbre de dérivation de cette grammaire qui permet d'obtenir les durées considérées.

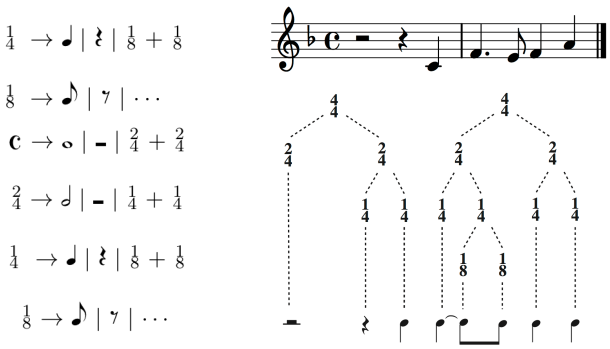

FIGURE 3.1 – [16] Un rythme et l'arbre de dérivation correspondant selon la grammaire hors-contexte décrite à gauche : la structure de l'arbre de dérivation est comparable aux arbres de rythme utilisés dans ce stage.

3.1.2 Schéma de subdivision

Le schéma de subdivision décrit les subdivisions possible du segment temporel d'entrée. En ce sens, il décrit un grammaire hors contexte. A chaque niveau, plusieurs subdivisions peuvent être autorisées. Pour décrire cela, le schéma est donné sous forme de listes imbriquées. Un schéma est défini récursivement comme suit :

$$
\begin{aligned}
Schema &= AndList \\
AndList &= liste\ de\ OrList \\
OrList &= valeur\ ou\ liste\ de\ AndList
\end{aligned}
$$

Les éléments d'une *AndList* correspondent à des divisions successives, et les éléments d'une *OrList* correspondent à différents choix de subdivisions à une profondeur donnée. Les deux types de listes sont représentés par des listes simples, mais pour en faciliter la lecture, nous séparerons les éléments des *OrList* par des barres verticales : | . Plusieurs exemples de schémas sont donnés, ainsi que les subdivisions possibles auxquels ils correspondent :

- (3 2 2) **:** On divise le segment en 3, puis on redécoupe chaque subdivision en 2, puis à nouveau en 2.
- ((2|3) 2 2) **:** On a le choix entre les schémas (2 2 2) et (3 2 2)
- ((2|3) ((2 3) | ((3|5) 2))) : Pour la première division, on peut diviser soit en 2, soit en 3. Ensuite, on peut soit diviser en deux puis en 3, soit diviser en 3 ou en 5, puis en 2. Cela revient à avoir le choix entre les schémas (2 2 3), (2 3 2), (2 5 2), (3 2 3), (3 3 2) et (3 5 2).

Ainsi, l'exemple de la Figure 3.1 correspond à un schéma de subdivision égal à (2 2 2 2).
On a la propriété suivante :

L'ensemble des arbres de rythme compatibles avec un schéma de subdivision est un langage d'automate d'arbre

En effet, à partir d'un schéma, on peut construire une grammaire hors contexte. Par exemple, le schéma ((2|3) ((2 3)((3|5) 2))) permet de générer la grammaire hors contexte suivante :

$$
\begin{aligned}
racine &\rightarrow 2_1 \mid 3_1 \mid \bullet \\
2_1 &\rightarrow 2_2 \mid 3_2 \mid 5_2 \mid \bullet \\
3_1 &\rightarrow 2_2 \mid 3_2 \mid 5_2 \mid \bullet \\
2_2 &\rightarrow 3_3 \mid \bullet \\
3_2 &\rightarrow 2_3 \mid \bullet \\
5_2 &\rightarrow 2_3 \mid \bullet \\
2_3 &\rightarrow \bullet \\
3_3 &\rightarrow \bullet
\end{aligned}
$$

Ici, le symbole $\bullet$ signifie qu'on ne resubdivise pas, c'est le seul symbole terminal.

3.1.3 Automates d'arbres pondérés

On peut également attribuer des poids aux arbres, et définir ainsi un automate d'arbres pondéré [10]. Un automate d'arbre standard est une application associant à un arbre une valeur booléenne : vrai si l'arbre est reconnu, c'est à dire si l'arbre appartient au langage d'arbres considéré, faux sinon. Un automate d'arbre pondéré est une application qui associe à un arbre une valeur de poids dans un semi-anneau. La propriété de compositionnalité, qui dit qu'à partir de deux automates d'arbres, on peut construire un automate reconnaissant l'intersection des deux langages, s'étend au cas pondéré de la façon suivante :

A partir de deux automates d'arbres pondérés, on peut construire un automate retournant pour un arbre le produit des poids retournés par les deux automates.

Pour attribuer ces poids, on s'inspire du principe de décision bayésien décrit dans [6] :

$$P(rythme|entree) \propto P(entree|rythme)P(rythme)$$

Cette équation peut s'interpréter de la façon suivante :

$$Poids\ d'un\ arbre \propto (distance\ entre\ l'entrée\ et\ le\ rythme)(complexité\ du\ rythme)$$

Le poids d'un arbre est donc vu comme le produit d'une mesure de distance et d'une mesure de complexité. Le choix de ces mesures est déterminant, nous y reviendrons à la partie 3.3. Une fois ces mesures définies, le problème de la transcription revient à trouver l'arbre de poids minimal que l'on peut générer avec la grammaire définie au préalable.

3.1.4 Arbre de poids minimal

Trouver l'arbre de poids minimal est un problème facile conceptuellement, mais difficile calculatoirement, car l'ensemble des arbres possibles devient vite trop grand pour être parcouru entièrement lorsque le schéma de subdivision, et donc l'automate, deviennent grands. Cependant, des algorithmes ont été développés, à l'aide de techniques de programmation dynamique, pour évaluer de façon paresseuse les poids des arbres et retourner celui de poids le plus faible sans parcourir tout l'espace des arbres possibles. Nous décrirons en détail dans la partie 3.2 un algorithme, basé sur le fonctionnement des algorithmes décrits dans [12], adapté à des automates générés par des schémas de subdivision. Il permet d'énumérer les k arbres de poids les plus faibles que l'on peut obtenir à l'aide d'un automate pondéré. C'est un algorithme récursif qui calcule le poids d'un arbre

à partir de ceux de ses fils. L'hypothèse qui permet de commencer l'énumération sans avoir exploré l'ensemble des solutions est celle de la monotonie :

Soit un noeud a, ayant k fils notés a_1, a_2, ... , a_k, et w la fonction qui associe à un noeud son poids. Alors

$$\forall i \in [1..k], w(a'_i) > w(a_i) \Rightarrow w(a') > w(a)$$

où a' est le noeud ayant pour fils a_1, a_2, ... , a_{i-1}, a'_i, a_{i+1}, ... , a_k

Autrement dit, augmenter le poids d'un des fils fera augmenter le poids du père. Ainsi, on sait que l'arbre de poids minimum sera obtenu en choisissant à chaque noeud les fils de poids minimum.

3.2 Principe de l'algorithme d'énumération des meilleures solutions : l'algorithme k-best

L'algorithme k-best que nous avons développé permet d'énumérer les meilleures transcriptions des durées d'entrée, au sens des mesures évoquées à la partie 3.1.3. Il se déroule en deux étapes principales. D'abord, on construit l'automate pondéré, qui est décrit par une table de hachage indexée par les états, et ensuite, on calcule les k meilleures solutions pour cet automate. Une fois l'automate construit, on peut facilement, et sans grand coût computationnel, obtenir d'autres solutions. Dans cette partie nous détaillerons le fonctionnement de cet algorithme.

3.2.1 Entrées et sorties de l'algorithme

Divers paramètres sont donnés à l'algorithme en entrée :

- Une série d'instants, correspondant aux instants de début des événements (notes ou silences). Cette série est normalisée de sorte que le premier événement commence à 0 et le dernier finit à 1. On a donc un segment à l'intérieur duquel se trouvent des instants à quantifier. Le dernier élément ne sera pas pris en compte, il ne sert qu'à indiquer la fin du segment.
- Une liste, de même longueur que la série d'instants, indiquant si chaque instant correspond à une note (n) ou si il correspond à un silence (s). Cette liste n'intervient pas dans l'algorithme en lui-même, elle ne sera utile que pour la reconstruction de l'arbre de rythme à la fin de l'algorithme.
- Un schéma de subdivision, donnant les subdivisions successives de l'entrée possible. Le format de ce schéma est donné à la partie 3.1.2.

La construction de l'arbre revient ensuite à découper récursivement ce segment, et aligner les instants d'entrée à la borne de la subdivision la plus proche, cette borne étant dépendante du schéma de subdivision et du nombre de divisions successives.

Une série d'instants correspond à une unité musicale, typiquement une mesure. Cela implique qu'un prétraitement a été effectué pour découper le flux d'entrée en pulsations et en mesures. Nous reviendrons sur ces points à la partie 4.1.2.

En sortie, on renvoie k arbres, ordonnés par poids, le poids étant une mesure de qualité des solutions proposées. La définition de ce poids sera discutée à la section 3.3.

On peut donc poser le problème de la façon suivante : en entrée, on a n instants, notés $x_1, x_2, ..., x_n$, et en sortie, on renvoie k arbres de rythmes notés $t_1, t_2, ..., t_k$, auxquels correspondent k séries d'instants quantifiés, notées $y_{11}, y_{12}, ..., y_{1n},\ y_{21}, y_{22}, ..., y_{2n},\ y_{k1}, y_{k2}, ..., y_{kn}$.

3.2.2 Construction de l'automate

L'automate est décrit par une table de hachage indexée par les états, c'est-à-dire les noeuds de l'arbre. Les noeuds sont représentés par le chemin pour y arriver depuis la racine. Un chemin a la

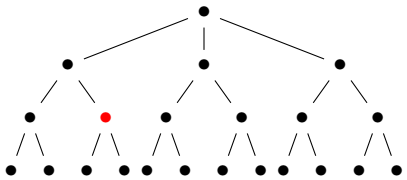

FIGURE 3.2 – Le noeud rouge est référencé par le chemin (3 1)(2 2)

forme suivante : $path = (a_1\ d_1)(a_2\ d_2)...(a_n\ d_n)$, où les a correspondent aux arités des noeuds, et les d indiquent la branche choisie pour arriver au noeud considéré. Un exemple est donné Figure 3.2. A chaque chemin correspond une portion du segment d'entrée : le chemin (3 1)(2 2) correspond ainsi à la deuxième moitié du premier tiers du segment.

3.2.2.1 Représentation interne des arbres

A aucun moment dans l'algorithme on ne manipule des arbres explicitement. Ils sont toujours donnés implicitement par récursion : on connait les fils de chaque noeud, mais pas l'arbre global (qui sera reconstruit uniquement à la fin, pour renvoyer un résultat). Les données manipulées sont appelées des "run", auxquels sont associés des poids. Il s'agit d'une liste d'entiers, de longueur égale à l'arité du noeud, et dont les éléments sont des indices dans la liste des k meilleurs résultats pour chaque état fils, c'est à dire la liste des k runs de poids les plus faibles de chaque état fils. Par exemple, pour un état décrit par un chemin p, un run valant (3 1 2) signifie que le sous-arbre considéré est obtenu en allant chercher le $3^{\text{ème}}$ meilleur run de l'état p(3 1), le meilleur run de l'état p(3 2) et le $2^{\text{ème}}$ meilleur run de l'état p(3 3). Un run égal à la liste vide signifie qu'on ne subdivise pas plus loin. Ainsi, en parcourant successivement les runs et les listes des meilleurs runs, on peut reconstituer l'arbre complet.

3.2.2.2 Format de l'automate

Chaque entrée de la table, qui est indexée par les états, contient :
— La liste des k meilleurs runs
— La liste des candidats parmi lesquels on va choisir le $k+1^{\text{ème}}$ meilleur run
— D'autre éléments stockés ici pour des raisons d'efficacité

A l'aide du schéma de subdivision, on construit les différents états, donc les différents chemins possibles. Par exemple, pour le schéma ((2|3) 2), on va construire les états rassemblés dans le tableau Figure 3.3.

Pour améliorer les performances de l'algorithme, on effectue quelques optimisations lors de la construction de l'automate. En effet, si le segment correspondant à un chemin ne contient aucune entrée, il est inutile de subdiviser plus finement, car il n'y a rien à aligner dans ce segment. On va alors se dispenser de créer les états fils de ce chemin, ce qui économisera des calculs à la fois à la création de l'automate et pendant la construction de l'arbre, puisqu'il y aura moins d'états à explorer.

Une fois l'automate créé, on initialise la liste des candidats. Pour cela, on place dans la table de hachage les premiers candidats ainsi que leurs poids. Ces premiers candidats sont :

— Le cas où on ne subdivise pas plus loin, on aligne directement les entrées qui se trouvent à l'intérieur du segment correspondant à l'état considéré aux bornes les plus proches. On place alors dans la liste des candidats un run vide et le poids, calculé à partir des distances entre les entrées et la borne la plus proche.
— On redécoupe en le nombre de subdivisions autorisées par le schéma. On place alors pour chaque subdivision possible le meilleur candidat qui est, d'après la propriété de monotonie,

Niveau 0	Niveau 1	Niveau 2
liste vide (*racine*)	(3 1)	(3 1), (2 1)
		(3 1), (2 2)
	(3 2)	(3 2), (2 1)
		(3 2), (2 2)
	(3 3)	(3 3), (2 1)
		(3 3), (2 2)
	(2 1)	(2 1), (2 1)
		(2 1), (2 2)
	(2 2)	(2 2), (2 1)
		(2 2), (2 2)

FIGURE 3.3 – Tableau recensant les états construits pour un schéma de subdivision ((2|3) 2). Les états sont classés par profondeur.

le run (1 1...1), où le nombre de 1 est égal à l'arité de la subdivision.

En reprenant l'exemple précédent, après l'initialisation, la liste des candidats de l'entrée de la table de hachage correspondant à la racine contiendra donc :

— Un run vide, et son poids initialisé avec la distance de chaque input à la borne du segment la plus proche (0 ou 1)
— Un run (1 1) et son poids qui n'a pas encore été calculé
— Un run (1 1 1) et son poids qui n'a pas non plus été calculé.

3.2.3 Déroulement de l'algorithme

Une fois l'automate créé, on peut dérouler l'algorithme. Il s'agit d'un algorithme récursif : on demande à chercher le $k^{\text{ème}}$ arbre de poids le plus faible en partant de la racine, et la fonction sera appelée récursivement sur tous les sous-arbres pour évaluer leurs poids. On note *best* la fonction récursive qui renvoie pour un certain état son $k^{\text{ème}}$ arbre de poids le plus faible. On va donc appeler $best(k, racine)$, qui retourne un arbre et son poids.

3.2.3.1 Une étape de l'algorithme

Supposons que l'on veuille retourner le $k^{\text{ème}}$ arbre (ou sous-arbre) de poids le plus faible, plusieurs cas sont possibles :

Cet arbre a déjà été évalué : Il suffit de le chercher dans la liste des meilleurs runs stockée dans la table de hachage à la ligne correspondant au chemin, et retourner son poids et le run correspondant.

Cet arbre n'a pas encore été évalué : On va chercher le run de poids le plus faible dans la liste des candidats pour la ligne correspondant au chemin :

— Si des poids n'ont pas été définis dans la liste des candidats, on applique récursivement l'algorithme pour les calculer. Par exemple, pour état p et un run (3 1 2), on appelle l'algorithme pour obtenir le poids du $3^{\text{ème}}$ meilleur run pour le 1^{er} fils ($best(3, p(3\ 1))$), celui du meilleur run pour le $2^{\text{ème}}$ fils ($best(1, p(3\ 2))$) et celui du $2^{\text{ème}}$ meilleur run pour le $3^{\text{ème}}$ fils ($best(2, p(3\ 3))$), puis on en déduit le poids du run (3 1 2).

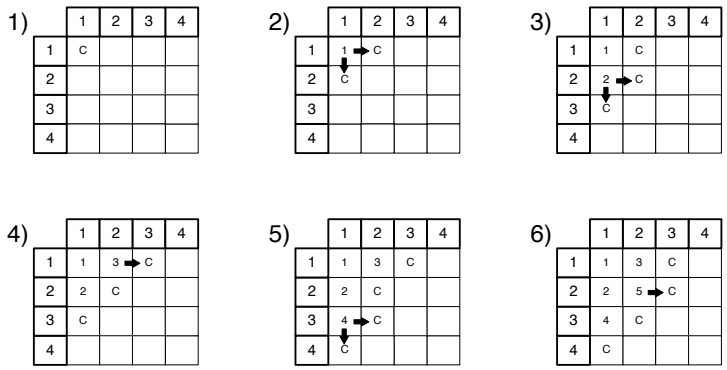

FIGURE 3.4 – Exemple d'ordre d'ajout des candidats pour un noeud d'arité 2 : La ligne donne le rang du run fils pour la branche de gauche, la colonne, celui de la branche de droite, C veut dire que le run est ajouté à la liste des candidats, les numéros donnent l'ordre d'ajout à la liste des candidats dans la liste des meilleurs runs.

— Si tous les poids des candidats ont été définis, on choisit celui de poids le plus faible, on le retire de la liste des candidats et on le place dans la liste des k meilleurs. On ajoute à la liste des candidats les prochains potentiels candidats, comme expliqué plus loin.

Si la liste des candidats est vide, par exemple dans le cas d'une feuille, où il n'y a qu'un candidat possible, ou dans le cas d'un noeud interne dont tous les candidats ont déjà été évalués et classés dans la liste des k meilleurs, on renvoie un valeur spéciale de poids $(+\infty)$. Ainsi, on évite de continuer à chercher lorsque toutes les possibilités ont déjà été énumérées.

Ainsi on va lancer l'algorithme à la racine, qui appellera récursivement l'algorithmes sur chaque sous-arbre, et ce jusqu'aux feuilles. Une fois arrivé aux feuilles, on peut commencer à évaluer les poids, et on évalue ainsi le poids de chaque sous arbre à partir du poids de ses fils.

3.2.3.2 Ajout des candidats suivants à la liste des candidats

Une fois qu'on a placé un candidat dans la liste des candidats, il faut déterminer qui peuvent être les suivants. La règle d'ajout est la suivante :

Soit un run $(i_1\ i_2\ ...\ i_n)$. On ajoute n nouveaux candidats de la forme :

$$((i_1+1)\ i_2\ ...\ i_n), (i_1\ (i_2+1)\ ...\ i_n), ..., (i_1\ i_2\ ...\ (i_n+1))$$

Cette règle découle de la propriété de monotonie. En effet, si par exemple on considère le meilleur run (1 1 ... 1), le deuxième meilleur run sera forcément parmi les suivants : (2 1 ... 1), (1 2 ... 1), ... , (1 1 ... 2) (à moins qu'une autre subdivision, déjà présente dans la liste des candidats, ne soit plus avantageuse). Un exemple de parcours des candidats possibles pour un noeud d'arité 2 est donné Figure 3.4. Les doublons ne sont pas pris en compte.

3.2.4 Un exemple simple

Nous allons dérouler l'algorithme dans un cas très simple pour en expliciter le fonctionnement. On prend comme fonction de poids une fonction naïve, définie pour une feuille par le produit de la somme des distances entre les entrées et la borne de la subdivision la plus proche et de la profondeur de l'arbre, et pour un noeud, par la somme des poids des fils.

Nous prenons comme entrée la série normalisée $(0,\ 0.45,\ 1)$, la liste $(n\ n\ s)$ indiquant que les deux premiers instants correspondent à des débuts de notes, et le schéma (2 2). Les entrées et les

subdivisions possibles sont représentées Figure 3.5. On peut d'ores et déjà supposer qu'une bonne quantification de ces durées serait le rythme ♪♪.

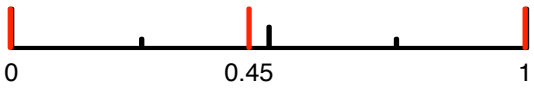

FIGURE 3.5 – L'entrée est représentée en rouge, les subdivisions possibles sont représentées en noir.

L'automate construit a donc les états suivants :

$$\begin{array}{c} \textit{liste vide}\ (\textit{racine}) \\ (2\ 1) \\ (2\ 2) \\ (2\ 1)(2\ 1) \\ (2\ 1)(2\ 2) \\ (2\ 2)(2\ 1) \\ (2\ 2)(2\ 2) \end{array}$$

Les listes de candidats et des meilleurs runs, stockés dans la table sous la forme de couples $(run, poids)$ sont initialisées comme suit :

Etat	Liste de candidats	Liste des meilleurs runs
liste vide (*racine*)	$(nil,\ 0.45 \times 1)\ ((1\ 1),\ undef)$	
(2 1)	$(nil,\ 0.05 \times 2)\ ((1\ 1),\ undef)$	
(2 2)	$(nil,\ 0 \times 2)\ ((1\ 1),\ undef)$	
(2 1)(2 1)	$(nil,\ 0 \times 3)$	
(2 1)(2 2)	$(nil,\ 0.05 \times 3)$	
(2 2)(2 1)	$(nil,\ 0 \times 3)$	
(2 2)(2 2)	$(nil,\ 0 \times 3)$	

On cherche à obtenir $best(1, racine)$. La liste des meilleurs runs est vide pour l'instant, on cherche donc dans la liste des candidats le meilleur. Un candidat dans la liste a un poids indéfini, on cherche donc à l'évaluer. Pour cela, comme le run non évalué vaut (1 1), on cherche $best(1, (2\ 1))$ et $best(1, (2\ 2))$.

Commençons par le fils de gauche. A nouveau, la liste des meilleurs runs est vide et le poids d'un des candidat n'est pas défini, on cherche donc à l'évaluer.

Comme le run non évalué vaut (1 1), on appelle $best(1, (2\ 1)(2\ 1))$ et $best(1, (2\ 2)(2\ 2))$. Pour l'état (2 1)(2 1), tous les candidats ont été évalués, celui de poids le plus faible est (nil, 0) : on le retire de la liste des candidats et on le place dans la liste des meilleurs runs. Désormais, lorsque l'on cherchera le meilleur run pour l'état (2 1)(2 1), il suffira d'aller le chercher dans la liste des meilleurs runs. Comme aucune subdivision n'est possible pour cet état, on ne rajoute pas de candidats, la liste des candidats reste vide. On fait de même pour l'état (2 1)(2 2).

A ce stade, la table de hachage est la suivante :

Etat	Liste de candidats	Liste des meilleurs runs
liste vide (*racine*)	(*nil*, 0.45) ((1 1), *undef*)	
(2 1)	(*nil*, 0.1) ((1 1), *undef*)	
(2 2)	(*nil*, 0) ((1 1), *undef*)	
(2 1)(2 1)		(*nil*, 0)
(2 1)(2 2)		(*nil*, 0.15)
(2 2)(2 1)	(*nil*, 0)	
(2 2)(2 2)	(*nil*, 0)	

On peut maintenant évaluer le run (1 1) de l'état (2 1) : son poids vaut $0+0.15 = 0.15$. Tous les candidats de cet état sont maintenant évalués, celui de poids le plus faible est le run correspondant au cas où on ne subdivise pas : on le retire de la liste des candidats et on le place dans la liste des meilleurs runs. Comme aucune subdivision n'est possible pour cet état, on ne rajoute pas de candidats.

De la même façon que pour l'état (2 1), on calcule le poids du run (1 1) de l'état (2 2) : son poids vaut $0 + 0 = 0$.

La table de hachage contient maintenant :

Etat	Liste de candidats	Liste des meilleurs runs
liste vide (*racine*)	(*nil*, 0.45) ((1 1), *undef*)	
(2 1)	((1 1), 0.15)	(*nil*, 0.1)
(2 2)	((1 1), 0)	(*nil*, 0)
(2 1)(2 1)		(*nil*, 0)
(2 1)(2 2)		(*nil*, 0.15)
(2 2)(2 1)		(*nil*, 0)
(2 2)(2 2)		(*nil*, 0)

On peut maintenant évaluer le run (1 1) de l'état racine. Son poids vaut $0.1 + 0 = 0.1$. A nouveau, on peut maintenant retirer le run de poids le plus faible de la liste des candidats de la racine, qui est le run (1 1). On le retire de la liste des candidats et on le place dans la liste des meilleurs runs, mais cette fois, on rajoute deux candidats pour cet état : (2 1) et (1 2).

A la fin de l'algorithme, la table de hachage contient :

Etat	Liste de candidats	Liste des meilleurs runs
liste vide (*racine*)	(*nil*, 0.45) ((2 1), *undef*) ((1 2), *undef*)	((1 1), 0.1)
(2 1)	((1 1), 0.15)	(*nil*, 0.1)
(2 2)	((1 1), 0)	(*nil*, 0)
(2 1)(2 1)		(*nil*, 0)
(2 1)(2 2)		(*nil*, 0.15)
(2 2)(2 1)		(*nil*, 0)
(2 2)(2 2)		(*nil*, 0)

Le résultat renvoyé est le meilleur run pour la racine : il s'agit du run (1 1) (sans surprises), avec le poids 0.1. Une fois l'arbre reconstitué, il correspond au rythme ♪♪, ce qui correspond bien à ce qu'on aurait voulu obtenir qualitativement.

Si on souhaite ensuite calculer le 2ème meilleur run, il suffit de relancer l'algorithme avec la même table de hachage.

3.3 Choix des mesures de poids

L'algorithme présenté précédemment est très générique : on aligne des instants sur une grille de pas variable, mais ces instants pourraient correspondre à n'importe quoi. Même au delà des instants, si on avait initialisé les feuilles avec d'autres valeurs de poids, pas forcément liées à des distances entre des instants, cet algorithme aurait pu être exploité pour résoudre d'autres problèmes : les algorithmes décrits dans [12] ont d'ailleurs été proposés pour le traitement du langage naturel. Le choix des mesures de poids en fait un algorithme spécifiquement destiné à la quantification rythmique.

Pour déterminer le poids d'un arbre, l'algorithme se base sur trois éléments : une mesure de précision, une mesure de complexité, et une fonction permettant de renvoyer un poids à partir de ces deux objets.

La mesure de précision est obtenue à partir d'un vecteur de dimension égale au nombre d'instants d'entrée. Le $i^{\text{ème}}$ élément de ce vecteur est égal à la différence entre le $i^{\text{ème}}$ instant d'entrée et le point de la grille sur lequel il est aligné. Autrement dit, en reprenant les notations de la section 3.2.1, le $i^{\text{ème}}$ élément de ce vecteur est égal à $|x_i - y_i|$. Pour un noeud interne de l'arbre, certains éléments de ce vecteur n'auront pas de valeur. En effet, nous avons vu à la partie 3.2.2 qu'à chaque noeud de l'arbre correspond une portion du segment d'entrée. Les x_i contenus dans cette portion seront alignés à la borne de la portion la plus proche, et on n'a aucune information sur les autres portions. La distance $|x_i - y_i|$ n'est donc connue que pour les x_i contenus dans cette portion.

La mesure de complexité est obtenue à partir de 3 valeurs :

- Un vecteur contenant le nombre d'occurences de chaque arité pour le sous-arbre considéré. Cette valeur permet de pénaliser les arités inhabituelles comme 5 ou 7.
- La profondeur du sous-arbre. Cette valeur permet d'éviter les subdivisions trop fines, qui donnent lieu à des rythmes peu lisibles
- Le nombre de notes alignées sur le même point de la grille. Cette valeur vise à éviter d'avoir dans l'arbre de sortie des "grace notes".

Reste à déterminer la fonction de poids combinant ces deux notions de précision et de complexité. Or, nous l'avons vu, la condition indispensable au bon déroulement de l'algorithme est celle de la monotonie de la fonction de poids : si on augmente le poids de l'un des fils, le poids du père doit aussi augmenter.

Pour déterminer le poids d'un arbre à partir de ses fils, on combine les précisions des fils d'une part et leurs complexités d'autre part pour obtenir la précision et la complexité du père, et on applique la fonction de poids à ces deux éléments pour obtenir le poids du père. Trouver une fonction de combinaison monotone simplement par rapport à la précision ou par rapport à la complexité est une tâche facile, trouver une fonction qui garantisse que si la combinaison de la précision et de la complexité d'un des fils augmente, la combinaison de la précision et de la complexité du père augmente aussi est beaucoup moins trivial.

On pose le problème de la façon suivante, dans le cas particulier d'un arbre ayant deux fils (il est facile de généraliser à n fils) :

Soient p_1, p_2, les précisions des fils (respectivement gauche et droit), c_1, c_2 leurs complexités respectives, p la précision du père et c sa complexité. Soient $w : c, p \longmapsto w(c, p)$ la fonction qui combine la précision et la complexité pour donner le poids, $f : c_1, c_2 \longmapsto f(c_1, c_2)$ la fonction qui

donne la complexité du père à partir de celle des fils et $g : p_1, p_2 \longmapsto g(p_1, p_2)$ la fonction qui donne la précision du père à partir de celle des fils. On veut que w, f et g vérifient :

$$Si\ w(c'_1, p'_1) > w(c_1, p_1),\ alors\ w(c', p') > w(c, p)$$

Autrement dit :

$$Si\ w(c'_1, p'_1) > w(c_1, p_1),\ alors\ w(f(c'_1, c_2), g(p'_1, p_2)) > w(f(c_1, c_2), g(p_1, p_2))$$

Nous avons choisi comme fonction de complexité une combinaison linéaire de tous les éléments évoqués plus haut : les distances point-à-point entre l'entrée et la grille (notées $d_1, d_2...d_n$), les nombres d'occurence des arités (notés $ca_1, ca_2...ca_m$, les p_j correpondent à des pénalités qui tépendent de la complexité de l'arité considérée), la profondeur de l'arbre (notée cd) et le nombre de notes alignées sur un même point de la grille (noté cg). On n'utilise que 4 coefficients pour cette combinaison linéaire :

$$poids = a_1 \times (\sum d_i) + a_2 \times (\sum (p_j ca_j)) + a_3 \times cd + a_4 \times cg$$

Ces 4 coefficients sont déterminés expérimentalement pour l'instant. Chaque coefficient détermine l'influence de chaque paramètre dans le résultat final, et donc si on privilégie plutôt les distances faible, les arités simples, les arbres peu profonds ou l'absence de "grace notes". Ces coefficients ont une grande influence sur les solutions obtenues, il faut donc envisager de les rendre réglables par l'utilisateur dans une certaine mesure.

Chapitre 4

Développements, évaluation et résultats

Nous avons présenté le travail formel et algorithmique réalisé pour la quantification rythmique. Nous allons dans cette partie présenter le travail technique d'implémentation qui a été réalisé et intégré dans OpenMusic et discuter les performances de l'algorithme, tant en termes de complexité algorithmique que de qualité de la quantification.

4.1 Bibliothèque k-best

4.1.1 Implémentation modulaire

Nous avons implémenté l'algorithme présenté à la partie précédente. L'implémentation a été faite en Lisp, et les différents fichiers ont été regroupés dans une bibliothèque OpenMusic. Différentes structures de données ont également été définies pour faciliter son implémentation et surtout, son amélioration. En effet, l'algorithme a été implémenté de façon modulaire, avec des données encapsulées dans des structures de données plus ou moins complexes et un nombre restreint de fonctions partagées. En particulier, le coeur de l'algorithme est indépendant de la façon dont les différentes mesures de précision, de complexité et le poids sont calculées, et même de la façon dont on compare les poids. Seules des fonctions comme `weight-compare`, qui permet de comparer deux poids, ou `weight-addall`, qui permet d'obtenir le poids d'un arbre à partir de ces sous-arbres sont appelées. Le coeur de l'algorithme est indépendant de la définition de ces fonctions. Cela permet en particulier de pouvoir améliorer ces différentes fonctions sans avoir à modifier tout l'algorithme.

4.1.2 Mise en forme de l'entrée

Nous avons vu que l'algorithme prend en entrée seulement une suite d'instants. Pour mieux l'intégrer à l'environnement OpenMusic, nous avons développé des fonctions permettant de créer une entrée utilisable par l'algorithme à partir de suites de notes regroupées en `chordseq`.

Ces fonctions prennent en entrée les instants de début et les durées des notes du `chordseq` ainsi que le tempo et la signature (éventuellement une liste de tempos et de signature si ils varient). Précisons que l'algorithme développé est un algorithme de quantification, le tempo et la signature sont donc supposés connus. On découpe ensuite l'entrée en mesures, puisque l'on connait la durée d'une mesure à partir des tempos et des signatures. Lorsqu'une barre de mesure coupe une note, on la divise en deux, et on stocke dans une liste un booléen indiquant que la première note de la deuxième mesure doit être liée à la dernière note de la première mesure. Cette liste de booléens sera utilisée lors de la reconstruction de l'arbre de sortie : elle permettra d'indiquer, lorsque l'on

recollera les mesures après quantification, qu'il faut lier la dernière note de la mesure précédente à la première de la suivante. Un exemple est donné en Figure 4.1. On obtient ainsi des unités facilement manipulables par l'algorithme.

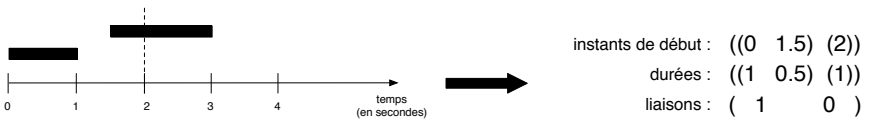

FIGURE 4.1 – L'entrée à gauche correspond à une liste d'instants valant (0 1.5) et une liste de durées valant (1 1.5). L'entrée est découpée en deux mesures de deux temps, avec un tempo de 60. Le 1 dans la liste "liaisons" indique que la dernière note de la première mesure doit être liée à la première note de la deuxième.

A partir de chaque mesure, on convertit la liste d'instants de débuts et la liste de durées en une suite d'instants, correspondant aux débuts et fins des notes, et une liste de même longueur indiquant si chaque instant correspond à un début de note ou à un début de silence. Ainsi, lorsque deux notes ne sont pas directement consécutives, on place dans la liste d'entrées 4 instants : l'instant de début de la première, l'instant de fin de la première, l'instant de début de la deuxième, et l'instant de fin de la deuxième, en précisant que l'instant de fin de la première correspond à un début de silence. Lorsque deux notes se superposent, on place dans la liste l'instant de début de la première et l'instant de début de la deuxième, en précisant que ces deux instants correspondent à des débuts de notes. La Figure 4.2 donne des exemples de mise en forme de l'entrée. Les instants sont normalisés de sorte que le premier instant vaut 0 et le dernier, qui correspond à la fin du segment, vaut 1.

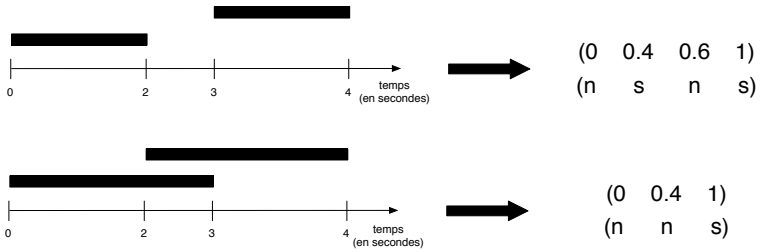

FIGURE 4.2 – Deux mesures différentes et leurs mises en formes. Dans les deux cas, le tempo vaut 60, et la signature est 5/4. n veut dire note, s veut dire silence. Les instants sont normalisés entre 0 et 1 pour être indépendants du tempo.

Une fois les mesures mises en forme, on les quantifie une par une à l'aide de l'algorithme précédemment décrit, et on les concatène ensuite, en liant les notes qui avaient été coupées lors du découpage en mesures.

4.1.3 Post-traitement

4.1.3.1 Construction de l'arbre de rythme

Comme nous l'avons évoqué à la partie 3.2.2.1, l'algorithme ne renvoie pas explicitement des arbres. Les arbres de rythme doivent être reconstruits récursivement à partir de la table de hachage, en partant de la racine et en allant jusqu'aux feuilles.

On nomme la fonction qui permet de reconstruire le $k^{\text{ème}}$ meilleur arbre pour un état : $build(k, etat)$. Pour reconstruire le $k^{\text{ème}}$ arbre, on appelle donc $build(k, racine)$.

FIGURE 4.3 – Arbre reconstruit par l'algorithme avec l'exemple de la partie 3.2.4

Si il n'est pas vide, c'est qu'on subdivise encore, auquel cas on appelle récursivement la fonction de reconstruction pour chacun des fils afin d'obtenir les sous-arbres indiqués par les runs.

Si il est vide, c'est qu'on ne resubdivise plus. On aligne donc les entrées contenues dans le segment à la borne du segment la plus proche. De nombreux sous-cas se posent alors, que nous ne détaillerons pas. L'idée générale est la suivante :

— Les entrées alignées à gauche sont prises en compte dans la subdivision courante. Si il y en a plusieurs, on ajoute donc des "grace notes" à la subdivision courante.
— Les entrées alignées à droite seront prises en compte à la subdivision suivante : elles seront placées en tant que "grace notes" dans la subdivision suivante.
— Si le segment correspondant à l'état courant ne contient aucune entrée, cela correspond à une liaison avec la section précédente (ou un silence si la section précédente se terminait sur un silence).

Pour illustrer, nous reprendrons l'exemple de la partie 3.2.4. On appelle $build(1, racine)$. Le meilleur run est (1 1), on appelle donc $build(1, (2\ 1))$ puis $build(1, (2\ 2))$ (ici, l'ordre d'appel est important, puisque des entrées de la subdivision (2 1) peuvent être déplacés dans la subdivision (2 2) si elles sont alignées à droite du segment).

Le meilleur run de l'état (2 1) est vide, on ne subdivise pas plus loin. Le segment contient deux entrées : la première et la deuxième. La première est recalée à gauche du segment, la deuxième à droite. On place donc une note dans la subdivision, et on retient d'une part que la deuxième entrée est recalée à droite pour pouvoir la prendre en compte dans le segment suivant, d'autre part que la dernière entrée de la subdivision correspond à une note et non à un silence pour savoir si il faut faure une liaison ou non avec la subdivision suivante dans le cas où elle ne contient aucune entrée.

Le meilleur run de l'état (2 2) est vide aussi, on ne subdivise pas plus loin. Le segment contient une seule entrée : la dernière, qui n'est pas prise en compte, elle ne sert qu'à indiquer la fin du segment à quantifier. Le segment ne contient donc aucune entrée à recaler. Comme le segment précédent avait une entrée alignée à droite, on place cette note dans la subdivision.

L'appel à $build(1, (2\ 1))$ renvoie une note, et l'appel à $build(1, (2\ 2))$ également. $build(1, racine)$ retourne donc l'arbre de la Figure 4.3, ce qui correspond avec la notation OpenMusic à l'arbre (1 (1 1)). On ajoute ensuite la signature pour obtenir le résultat voulu.

4.1.3.2 Format des résultats

Les résultats sont présentés sous la forme d'une liste ordonnée d'arbres de rythme correspondant aux k meilleures transcriptions possibles. La première transcription est obtenue en concaténant les meilleurs arbres de toutes les mesures, la deuxième, en concaténant les deuxièmes meilleurs résultats de toutes les mesures, et ainsi de suite. L'utilisateur peut ensuite visualiser chacune de ces transcriptions dans un objet `voice`, ou bien toutes à la fois dans un objet `poly`. Il peut ainsi choisir pour chaque mesure la transcription qu'il préfère. Pour cela, il suffit de chercher dans la liste des résultats les mesures qu'il souhaite garder et les concaténer dans une nouvelle liste.

4.1.4 Grace notes

Comme nous l'avons vu à la partie 4.1.3.1, lors de la construction de l'arbre de rythme, des "grace notes" peuvent être insérées dans l'arbre. Il s'agit d'une extension nouvelle du formalisme d'arbre de rythme. Le format est le suivant : lorsqu'un arbre de la forme $(D(Subdivisions))$ a au

moins un zéro dans sa liste *Subdivisions*, alors il sera considéré comme une note accompagnée de grace notes. La durée de cette note est donnée par D, et le nombre de grace notes est donné par le nombre de 0 dans *Subdivisions*. Par exemple,l'arbre (1(0 1 0 0)) est considéré comme une note de durée 1, à laquelle sont attachées une grace note avant la note et deux après.

Dans OpenMusic, la représentation graphique des grace notes n'est pas encore gérée. Une note à laquelle une grace note est attachée va être considérée comme un accord de deux notes, où la deuxième note aura un léger décalage temporel (*offset*), positif ou négatif en fonction de la position de la grace note par rapport à la note principale. Un exemple de représentation dans OpenMusic d'une note avec des grace notes est donné Figure 4.4.

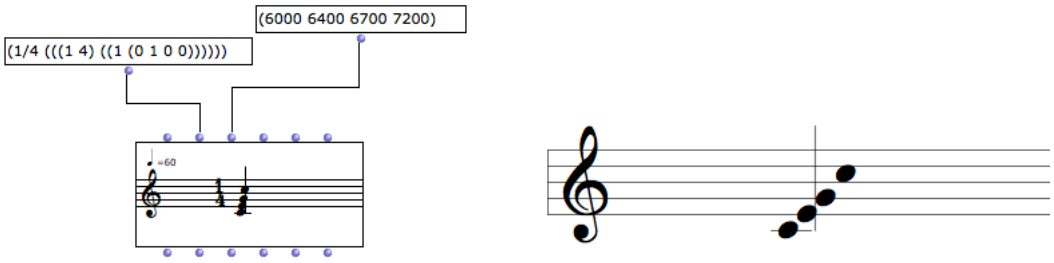

FIGURE 4.4 – Représentation des grace notes dans OpenMusic. A gauche, un exemple d'arbre de rythme permettant de représenter les grace notes est présenté. A droite, l'accord correspondant à l'arbre de gauche. On voit que la note tombant sur le temps est la deuxième, il y a une grace note avant et deux après.

4.2 Développements pour OMrewrite

La bibliothèque OMrewrite inclue des fonctions permettant la conversion d'arbres de rythme d'OpenMusic vers le format d'arbres de rythme symboliques, des fonctions pour faire de la réécriture d'arbres et un prototype de quantificateur par réécriture. Nous avons également réalisé des développements pour cette bibliothèque. Nous avons éliminé certaines erreurs qui se produisaient lors de la conversion d'arbres de rythmes d'OpenMusic vers les arbres de rythme symboliques. Nous avons également amélioré le système de quantification par réécriture, en rendant plus flexible la création de l'arbre de profondeur maximale. Désormais, ce dernier peut être construit à partir d'un schéma de subdivision simple, décrivant les subdivisions successives à la manière de l'algorithme k-best, mais en interdisant les *OrList*. On ne laisse pas de choix dans les subdivisions, on décrit simplement les subdivisions successives. Ainsi, le schéma (3 2 4) donnera l'arbre représenté Figure 4.5. La gestion des signatures rythmiques a également été améliorée, en permettant de les manipuler explicitement (dans le système précédent, une heuristique permettait de la deviner, souvent avec des erreurs).

Cette librairie est très intéressante utilisée en complément de l'algorithme k-best développé. En effet, elle permet d'obtenir des reformulations des rythmes obtenus. En particulier, elle permet de simplifier les solutions où l'algorithme a subdivisé inutilement un segment ne contenant qu'une note. Ce cas est fréquent, comme nous le verrons plus tard. On peut ainsi soit garder toutes les solutions, même celles comportant des subdivisions inutiles, ou bien au contraire réécrire ces solutions pour ne garder que celles qui sont vraiment différentes les unes des autres.

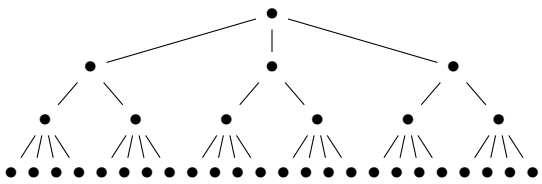

FIGURE 4.5 – Cet arbre est généré à l'aide de la liste de subdivisions (3 2 4)

4.3 Analyse de complexité de l'algorithme k-best

On utilise les notations suivantes :

a_{max} : Arité maximale que l'on peut trouver dans le schéma de subdivision
d_{max} : Profondeur maximale du schéma de subdivision (correspond à la longueur cumulée des *AndLists*)
c_{max} : Nombre maximum de choix d'arité qu'on a pour une subdivision
k : Nombre de solutions que l'on demande à l'algorithme de renvoyer
t : Durée totale du flux d'entrées à quantifier
n : Nombre d'instants contenus dans une mesure à quantifier

Dans cette section, nous allons évaluer la complexité en temps des différentes étapes de l'algorithme k-best. La complexité sera donnée dans le pire des cas, non sans discuter des cas les plus courants, et nous détaillerons quelques exemples.

4.3.1 Mise en forme de l'entrée

La mise en forme de l'entrée se fait en découpant le flux d'entrée en mesures et en créant à partir de chaque mesure un objet utilisable par l'algorithme. Pour créer un objet utilisable par l'algorithme, il faut également calculer la résolution du schéma, ce qui se fait en calculant le plus petit multiple commun des produits des arités de chaque schéma de subdivision possibles.

Le découpage en mesures se fait en temps linéaire avec le nombre de mesures, et donc la durée totale du flux à quantifier. La création de chaque entrée se fait en temps linéaire avec le nombre d'instants contenus dans la mesure, et le calcul de la résolution du schéma se fait en temps linéaire avec le nombre de schémas de subdivision possibles, soit au pire $d_{max}c_{max}$. La complexité de la mise en forme de l'entrée est donc en $O(t(n + d_{max}c_{max}))$.

4.3.2 Construction de l'automate

La construction de l'automate se fait en temps linéaire par rapport au nombre d'états, c'est-à-dire au nombre d'entrées de la table de hachage. Le nombre d'entrées est déterminé par le schéma de subdivision, il peut être borné grossièrement par $(a_{max}c_{max})^{d_{max}}$. Plus précisément, pour obtenir le nombre de chemins possibles (et donc le nombre d'états), il faut multiplier les éléments des *AndLists* et additionner ceux des *OrLists*. Ainsi, le schéma de subdivision ((2|3) ((4 5)|(5 4))), qui est équivalent à donner le choix entre les schémas (2 4 5), (2 5 4), (3 4 5) et (3 5 4), donne lieux à $(2 + 3) \times ((4 \times 5) + (5 \times 4)) = 200$ chemins (dans ce cas, $(a_{max}c_{max})^{d_{max}} = (5 \times 2)^3 = 1000$).

Notons que dans la pratique, le nombre d'états effectif sera souvent inférieur au nombre de chemins possibles. En effet, pour accélérer le traitement, on ne construit pas les états correspondant à des subdivisions d'une portion du segment à quantifier qui ne contient pas d'entrées. Lorsqu'on a peu d'entrées par rapport à la granularité permise par le schéma de subdivision, cette optimisation

peut faire chuter grandement le nombre d'états dans l'automate. Par exemple, si on considère le schéma précédent, et une entrée ne contenant qu'un seul instant à quantifier situé au début du segment, alors le deuxième et le troisième tiers (à fortiori la deuxième moitié) du segment sont vides. Tous les chemins commençant par (2 2), (3 2) ou (3 3) ne seront donc pas créés, ce qui fait chuter le nombre d'états à $(1 + 1) \times ((4 \times 5) + (5 \times 4)) = 80$.

Ensuite, à chaque état, l'initialisation des listes de candidats se fait en temps linéaire par rapport à c_{max}, puisqu'on place dans chaque liste un candidat pour chaque arité possible (une liste ne contenant que des 1 de longueur égale à l'arité) plus le candidat correspondant au cas où on ne subdivise plus.

La complexité de cette étape est donc en $O(c_{max}(a_{max}c_{max})^{d_{max}})$

4.3.3 Obtention des k meilleurs arbres

On distingue deux cas : la construction du meilleur arbre, et la construction des $k-1$ suivants.

4.3.3.1 Meilleur arbre

Pour obtenir le meilleur arbre une fois l'automate construit et initialisé, il faut d'abord évaluer les poids de tous les candidats pour chacun des états (puisque aucun poids n'a encore été évalué). On procède en commençant par les feuilles (la liste de candidats ne contient alors que le candidat correspondant au cas où on ne subdivise plus, dont le poids est directement calculé), et à chaque noeud, on remonte en évaluant le poids du candidat à partir de celui des fils. Le calcul du poids d'un père est linéaire en fonction du nombre de fils. On peut ainsi calculer en une passe le poids de tous les candidats des feuilles jursqu'à la racine.

En considérant le pire cas, on a donc $(a_{max}c_{max})^{d_{max}}$ états, pour lesquels il faut calculer c_{max} poids, chacun à partir de a_{max} fils. La complexité de cette étape est donc en $O((a_{max}c_{max})^{d_{max}+1})$.

Cette complexité est beaucoup sur-évaluée, d'une part parce que le nombre d'états est sur-évalué (nous l'avons vu à la partie précédente), et d'autre part parce qu'il est impossible que tous les candidats de chaque liste soient tous de longueur a_{max}, et on n'aura pas forcément c_{max} candidats à chaque étape non plus.

4.3.3.2 Construction des k-1 arbres suivants

Une fois que les candidats initiaux ont été évalués, la situation est différente. A chaque fois que pour un chemin donné, un run est ajouté à la liste des meilleurs arbres depuis la liste des candidats, on rajoute à la liste des candidats autant de candidats que l'arité du run ajouté, c'est-à-dire dans le pire des cas a_{max}. Chacun des runs sera identique à celui qui a été ajouté, sauf un des fils qui sera différent (cf. partie 3.2.3.2). Pour évaluer le poids de chacun de ces nouveaux candidats, il n'y aura donc qu'un seul poids fils à évaluer, celui du fils qui est différent, puisque les poids des fils qui sont identiques ont déjà été calculés lors de l'évaluation du poids du candidat qui vient d'être ajouté à la liste des meilleurs arbres. Par exemple, si on considère que le run (1 1 1) vient d'être ajouté à la liste des meilleurs runs, 3 nouveaux candidats sont ajoutés à la liste : (2 1 1), (1 2 1) et (1 1 2). Pour évaluer de chacun de ces runs, il suffira de calculer les deuxième meilleurs arbres pour chacun des fils, puisque les premiers ont déjà été calculés pour obtenir le poids du run (1 1 1) : il n'y a donc que 3 poids à évaluer.

On aura donc à chaque candidat au pire a_{max} poids à évaluer, chacun appartenant à une ligne différente de la table de hachage (chacun des a_{max} poids à évaluer correspond à un élément différent du run, et donc à un fils différent). Pour évaluer un de ces poids, il faudra évaluer tous les poids non évalués de la liste de candidats correspondante. Or, d'après le raisonnement précédent, on sait que cette liste de candidats contiendra au pire a_{max} candidats dont le poids n'est pas évalué, et pour chacun de ces candidats, il y a au pire 1 fils à évaluer. Sachant que pour les feuilles, il n'y

a qu'un seul candidat possible, donc aucun poids à évaluer, et qu'à la racine, il y a au plus a_{max} poids à évaluer, il y a donc au plus $a_{max}^{d_{max}}$ poids à évaluer.

Le calcul du poids d'un arbre à partir du poids de ses fils se fait en temps proportionnel au nombre de fils, donc en temps proportionnel à a_{max}. Une fois les poids calculés, il faut classer le poids du candidat qui vient d'être calculé dans la liste des candidats. La liste des candidats contient dans le pire des cas de l'ordre de ka_{max} candidats, puisque à chaque étape, on rajoute a_{max} candidats. Comme on utilise une structure de tas pour la liste des candidats, le classement du poids du candidat se fait en $O(log(ka_{max}))$

Pour évaluer les $k-1$ arbres suivants, il faut donc effectuer $k-1$ fois les opérations décrites précédemment dans cette partie. La complexité de l'obtention des $k-1$ arbres suivants est donc dans le pire des cas en $O(k(a_{max}^{d_{max}} \times a_{max} \times log(ka_{max})))$.

En réalité, cette complexité sera plus faible. Au delà du fait que toutes les arités ne sont pas égales à a_{max}, on n'ajoute pas non plus a_{max} candidats à la liste des candidats à chaque étape. En effet, si par exemple, pour un arbre binaire, le run (2 1) vient d'être ajouté à la liste des meilleurs arbres, on ajoute les runs (3 1) et (2 2) à la liste des candidats. Si le meilleur arbre suivant est (1 2), cette fois, on n'ajoutera à la liste des candidats que (1 3), puisque (2 2) a déjà été ajouté à la liste des candidats et qu'on élimine les doublons.

De plus, tous les poids n'auront pas besoin d'être évalués. En effet, certaines valeurs de poids ne peuvent pas être évaluées (par exemple, le $2^{\text{ème}}$ meilleur arbre pour une feuille n'existe pas, la seule possibilité étant de ne pas resubdiviser et d'aligner les entrées à la borne la plus proche). Ces runs impossibles à évaluer sont stockés avec une valeur spéciale de poids ($+\infty$). Lorsque l'on souhaite évaluer le poids d'un run, et que pour l'un des fils, la meilleure possibilité est un run avec un poids $+\infty$, on sait qu'on ne pourra pas évaluer le poids du run père, on lui attribue donc directement le poids $+\infty$. Cela permet d'une part de ne pas évaluer les poids des autres fils, et de ne pas passer par l'étape de calcul du poids du père à partir de celui des fils.

La complexité de l'obtention des k meilleurs arbres est donc en :

$$O((a_{max}c_{max})^{d_{max}+1} + k(a_{max}^{d_{max}+1}log(ka_{max})))$$

Selon le schéma de subdivision et le nombre de solutions demandées, c'est donc plutôt la construction du premier arbre ou l'obtention des $k-1$ suivants qui prédominera. On constate aussi que le temps de calcul n'est pas proportionnel au nombre de solutions que l'on souhaite obtenir : il y a un coût initial assez fort, mais l'obtention de solutions supplémentaires est relativement peu couteuse.

4.3.4 Construction de l'arbre de sortie

La construction de l'arbre de sortie se fait en parcourant la table de hachage depuis l'état racine jusqu'aux feuilles. A chaque noeud, on ne fait aucun traitement, on regarde simplement dans la table de hachage si on resubdivise ou non, et dans le cas où on resubdivise, combien d'éléments sont alignés à gauche de la subdivision et à droite.

Dans le cas où on resubdivise, on appelle récursivement la fonction de construction de l'arbre sur les fils du noeud considéré, on peut donc négliger la complexité de ces étapes devant la complexité dans le cas d'une feuille. Il y a au pire des cas $a_{max}^{d_{max}}$ feuilles. La complexité de l'étape de construction de l'arbre est donc en $O(a_{max}^{d_{max}})$.

Notons que cette étape est réalisée pour chaque solution possible, la construction de k arbres se fait donc en $O(ka_{max}^{d_{max}})$

4.3.5 Exemples

Nous avons réalisé quelques test de rapidité d'exécution de l'algorithme que nous avons implémenté. Les tests ont été réalisés sur un ordinateur iMac sous le système d'exploitation OS X

N	n	Schéma	E	k	t (en s)
1	5	A	31	10	0.325
				20	0.538
		B	139	10	0.609
				20	0.829
	10	A	41	10	0.392
				20	0.615
		B	159	10	0.684
				20	0.995
5		A	41	10	1.939
				20	3.043
		B	159	10	3.407
				20	4.959

FIGURE 4.6 – Résultats pour une entrée d'une seule mesure

10.10, équipé d'un processeur Intel Core i5 3,8 GHz et d'une mémoire vive de 4 Go. Pour mesurer le temps d'exécution, nous avons utilisé la fonction Lisp `get-internal-real-time`, appelée au début et à la fin de l'exécution de l'algorithme de quantification. Les tests ont été réalisés avec différents paramètres, les résultats sont rassemblés dans le tableau Figure 4.6. Dans ce tableau, les colonnes sont :

N : Nombre de mesures à quantifier

n : Nombre de notes à quantifier dans chaque mesure

Schéma : Le schéma de subdivision utilisé. Le schéma A est un schéma simple correspondant à une mesure à 4 temps binaire : (4 2 2 2). Le schéma B est un schéma plus complexe, permettant différentes subdivisions : (((2|3) (2|3) 2) | (5 (2|3) 2) | ((7|11|13))). Il est équivalent à laisser le choix entre les schémas ((2|3) (2|3) 2), (5 (2|3) 2), et subdiviser une seule fois en 7, en 11 ou en 13.

E : Nombre d'états dans l'automate (dépend uniquement du schéma de subdivision et des instants à quantifier)

k : Nombre de solutions demandées

t : Temps de calcul des solutions (en s).

Ces quelques résultats nous permettent de vérifier un certain nombre de comportements de l'algorithme expliqués à la partie précédente :

— La taille de l'automate est importante pour la vitesse d'exécution de l'algorithme. Le temps d'exécution de l'algorithme pour le schéma A est presque deux fois plus court que pour le schéma B. En outre, lorsque l'on rajoute des notes, on rajoute des états, et on voit, en comparant les résultats pour 5 notes et pour 10 notes, que cela a une petite influence sur le temps d'exécution. Il faudra donc veiller à utiliser le schéma le plus petit possible, et ne pas inclure de subdivisions dont on sait à l'avance qu'ont ne veut pas les obtenir.

— Une fois l'automate créé, retourner d'autres solutions est effectivement moins coûteux : le temps mis pour obtenir 20 solutions n'est pas égal au double du temps mis pour obtenir 10

solutions. On constate également que le temps mis pour obtenir les 10 solutions supplémentaires dépend peu de l'automate. On peut en déduire que le fait d'avoir un gros automate est pénalisant surtout à la construction de l'automate et à l'initialisation, et assez peu lors de l'obtention de solutions supplémentaires.

— Quantifier 5 mesures prend autant de temps que de quantifier 5 fois une seule mesure. En effet, à chaque nouvelle mesure, on reconstruit un nouvel automate, et on repart donc de zéro.

— Cet implémentation est plutôt longue à exécuter (5 secondes pour obtenir 5 mesures de partition). Il faudrait trouver des moyens d'accélérer les traitements. Des solutions seront proposées au chapitre 5.2.

4.4 Résultats

4.4.1 Exemples de résultats

Un exemple de patch est donné en Figure 4.7. Des exemples de résultats sont donnés aux Figures 4.8 à 4.11. A chaque fois, on donne l'entrée (donnée sur la première ligne), et les 10 meilleures transcriptions selon l'algorithme. Sauf indication contraire, l'entrée est quantifiée en prenant des mesures à quatre temps, et un tempo de 60 à la noire. Sur l'entrée, les positions théoriques des temps sont marquées en pointillés.

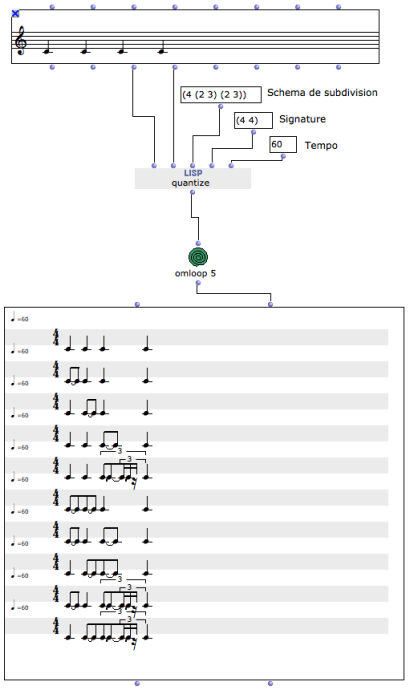

FIGURE 4.7 – Un exemple de patch pour quantifier une série de notes. Ici, la boucle `omloop` ne sert qu'à placer les différentes propositions dans un objet `poly`.

A partir de ces quelques résultats, on peut faire plusieurs constats :

Influence du schéma de subdivision : Le schéma de subdivision influe grandement sur les résultats possibles. Si l'on a une idée des résultats que l'on souhaite obtenir, en particulier

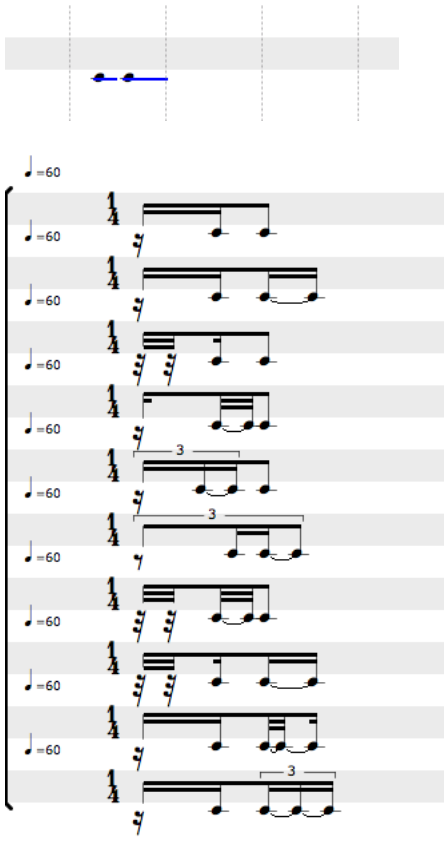

FIGURE 4.8 – Ici on ne quantifie qu'un temps. Le schéma utilisé est (((2|3) (2|3) 2) | (5 (2|3) 2) | ((7|11|13))). Les traits bleus indiquent les durées des notes.

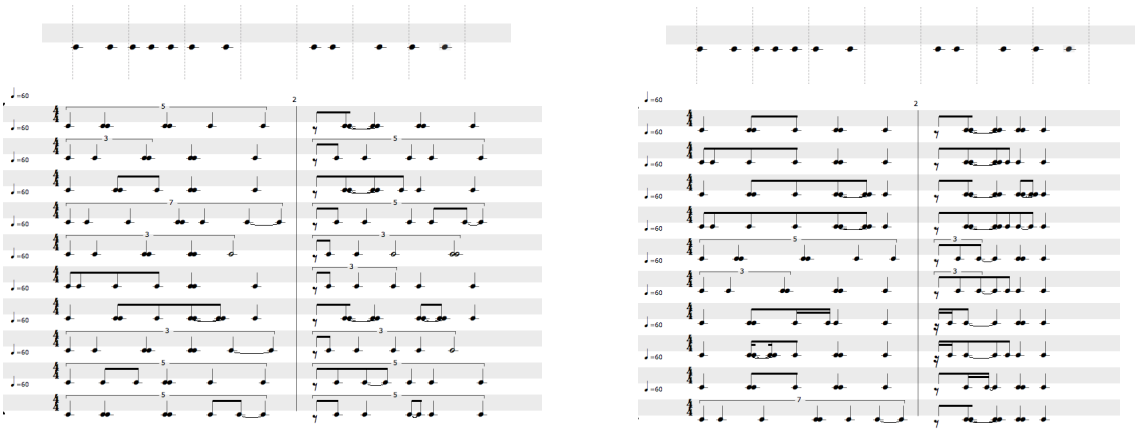

FIGURE 4.9 – Une même entrée quantifiée avec deux schémas différents : à gauche, (((2|3) (2|3) 2) | (5 (2|3) 2) | ((7|11|13))), à droite, (((2|3) (2|3) 2) | (4 (2|3) 2) | ((5|7|11|13))).

si l'on ne souhaite avoir que des rythmes binaires, utiliser un schéma adapté garantit l'ob-

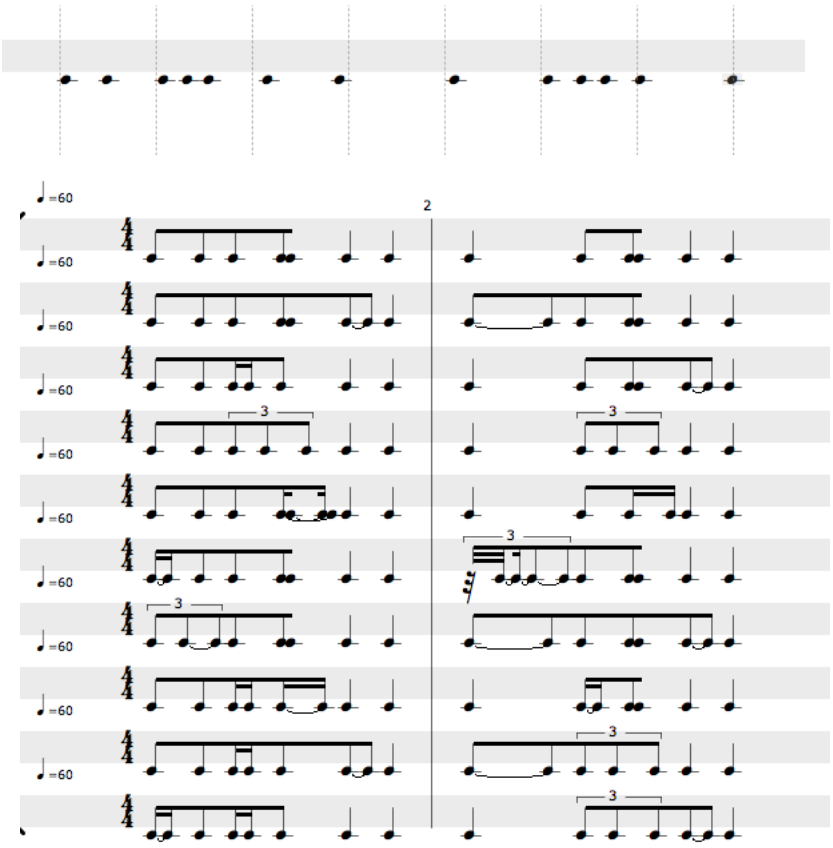

FIGURE 4.10 – Le schéma utilisé est (4 (2|3) (2|3)). Ici, une quantification qui semblerait logique serait de prendre la 3ème proposition pour la première mesure et la 4ème pour la deuxième mesure.

tention de résultats correspondant aux attentes, comme par exemple dans la Figure 4.10. Au contraire, si l'on n'a aucun à priori sur la forme des résultats que l'on souhaite obtenir, il peut être intéressant de prendre un schéma offrant de nombreuses possibilités, mais on risque d'obtenir des rythmes complexes (en particulier dans le cas de subdivisions par 5 ou 7), comme on peut le voir dans la partie gauche de la Figure 4.9. On peut également régler l'influence du paramètre de complexité relatif à l'arité des subdivisions pour limiter les occurences de telles subdivisions.

Grace notes : On remarque que les grace notes sont plutôt fréquentes dans les exemples considérés, en particulier dans les exemples de la Figure 4.9. Là encore, un meilleur réglage du paramètre de complexité relatif aux grace notes permettra d'empêcher qu'elles ne soient trop fréquentes. Notons cependant qu'on ne pourra jamais les empêcher complètement, même en rendant ce paramètre complètement discriminant. En effet, c'est lors de la construction de l'arbre, c'est-à-dire à la toute fin de l'algorithme, que l'on sait effectivement si il y a des grace notes ou non. Avant cela, on ne peut pas en être sûr : si plusieurs entrées correspondant à des notes sont alignées à la même extrémité d'une subdivision, on est sûr qu'il y aura des grace notes, mais si une entrée correspondant à une note est alignée à droite, elle peut donner lieu à une grace note si le segment suivant a une entrée correspondant à une note alignée à gauche, ou non, dans le cas contraire.

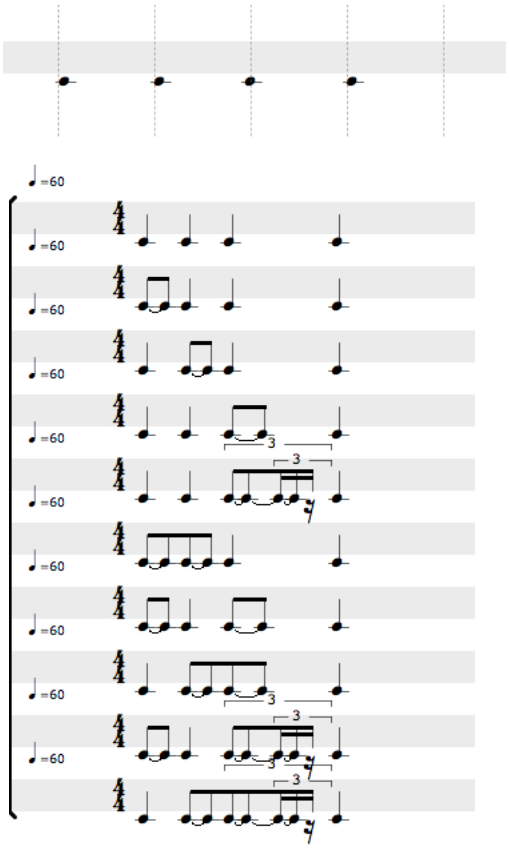

FIGURE 4.11 – On quantifie quatre noires avec peu d'imprécision. Le schéma utilisé est (4(2|3)(2|3)). Seules les rythmes 5, 9 et 10 correspondent à des durées différentes du premier.

Liaisons : On remarque que souvent, parmi les solutions proposées, certaines sont équivalentes, c'est à dire qu'une fois jouées, elles donnent exactement les mêmes durées. Par exemple, dans la Figure 4.11, la deuxième proposition est la même que la première, sauf que la première noire a été remplacée par deux croches liées, ce qui vaut la même durée. L'automate a été construit pour éviter certaines de ces subdivisions inutiles, en ne subdivisant pas des segments vides. Ici, les segments ne sont pas vides, ils contiennent chacun un instant à quantifier. L'algorithme va donc proposer malgré tout ces solutions redondantes, car en re-subdivisant, il est possible que l'instant contenu dans le segment ne soit alors plus aligné au même endroit. Ce n'est en fait qu'à la reconstruction de l'arbre que l'on constate que les solutions sont équivalentes. On peut adopter deux attitudes face à ces solutions redondantes. Soit elles peuvent être vues comme des doublons, auquel cas, on va chercher à s'en débarasser, soit elle peutvent être vues comme d'autres façons valides de noter le rythme d'entrée, auquel cas on les conserve, et on laisse le choix à l'utilisateur d'adopter l'une ou l'autre. Nous avons pris le deuxième parti dans cette implémentation.

Un exemple où notre algorithme montre ses limites est le suivant : on cherche à quantifier avec un tempo de 60 à la noire une note très courte, d'une durée de 100 ms, commençant au début du temps, suivie d'un silence. Les résultats renvoyés par notre algorithme sont donnée en Figure 4.12. On voit que la meilleure transcription ne tient pas compte du silence. Cela vient du fait que comme l'instant de début de la note et l'instant de début du silence sont tous les deux proches de la borne

gauche du segment, l'algorithme estime qu'il est moins couteux d'aligner les deux instants à gauche du segment sans subdiviser du tout. Ce choix est justifiable : si les deux instants correspondaient à deux débuts de notes, à la reconstruction de l'arbre, on obtiendrait une note à laquelle est attachée une grace note, ce qui serait une notation tout à fait satisfaisante pour ce genre de rythme.

Le problème vient en fait de la reconstruction de l'arbre de rythme. La notation qui correspondrait à cet alignement serait une grace note attachée à un silence. Or cette notation n'existe ni dans la théorie musicale, ni dans le formalisme utilisé dans OpenMusic. Ne pouvant pas noter une grace note accrochée à un silence, nous reconstruisons l'arbre en estimant que la note prend toute la durée de la subdivision, plutôt que de considérer que c'est le silence qui l'emporte. Cela revient en fait à aligner le début du silence à droite de la subdivision, et non à gauche, comme cela a été fait pour le calcul de la distance (alignement au plus proche). Il y a donc une différence entre le résultat renvoyé par l'algorithme et l'arbre reconstruit, ce qui est un véritable problème. Une solution à cet aspect sera proposée à la partie 5.2.2.2.

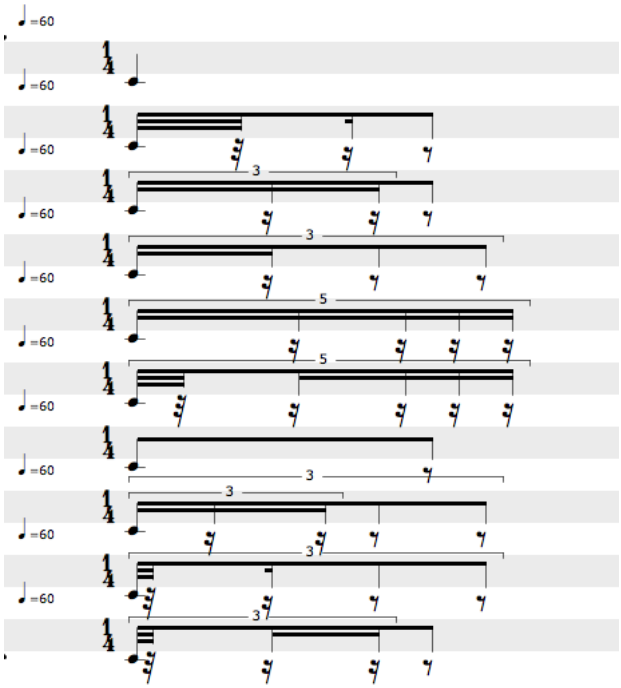

FIGURE 4.12 – On quantifie une note courte suivie d'un silence. Le schéma de subdivision utilisé est (((2|3) (2|3) 2) | (5 (2|3) 2) | ((7|11|13))). Le meilleur résultat ne tient pas compte du silence.

4.4.2 Comparaisons avec `omquantify`

La Figure 4.13 montre les résultats obtenus à l'aide de la fonction `omquantify` d'OpenMusic pour les mêmes entrées que celles étudiées à la partie 4.4.1. On réalise tout d'abord des tests avec les paramètres par défaut d'`omquantify` : toutes les subdivisions inférieures à 8 sont autorisées, et le paramètre de précision, qui permet de privilégier plus ou moins les notations précises par rapport aux notations complexes, est réglé à 0.5. On constate que pour beaucoup d'entre elles, on obtient des rythmes difficiles à lire. Dans le premier exemple, on obtient un septolet là où l'autre algorithme proposait plutôt des divisions simples, en deux ou en trois. Pire encore, une note est éliminée. Cela provient du fait que la fonction `true-durations`, utilisée pour mettre en forme les durées, a inséré

un court silence entre les deux notes. `omquantify` va donc chercher à la fois à quantifier le court silence et la note suivante, chose qu'il ne peut pas faire avec les contraintes imposées (on découpe le temps au maximum en 8 parties égales). La fonction va donc éliminer une des deux durées, et c'est la note qui est éliminée plutôt que le silence. Dans le deuxième exemple, on obtient à nouveau beacoup de septolets, difficilement lisibles. Notre algorithme, quant à lui, renvoie une solution contenant beaucoup de grace notes, ce qui n'est pas non plus idéal puisque très peu précis. Dans le troisième exemple, si certains temps sont quantifiés de façon vraisemblable, on aurait par exemple préféré que le premier temps soit représenté par deux croches que par un septolet. Encore une fois, notre algorithme va préférer avoir recours aux grace notes. Dans le quatrième exemple, le décalage de 60 ms (très faible, donc) de la troisième note est retranscrit dans la notation en un décalage d'une triple croche. Ces notations sont difficiles à lire, alors que les différences avec des notations plus simples, par exemple certaines notations renvoyées par notre algorithme, sont minimes.

On ajuste à présent les différents paramètres de `omquantify` pour obtenir des notations plus vraisemblables, et plus en accord avec les schémas de subdivisions utilisés (toutes les subdivisions ne sont pas autorisées par nos schémas). Les résultats sont affichés Figure 4.14. Ainsi, en interdisant les septolets, on obtient de meilleurs résultats dans tous les exemples. Sur le premier exemple, cela permet d'obtenir la bonne figure rythmique, mais à cause du silence rajouté par `true-durations`, on a un demi soupir à la place de la dernière croche. Pour le deuxième exemple, on interdit les septolets et on règle le paramètre de précision à 0.1 pour obtenir un rythme assez simple. Pour le troisième exemple, interdire les septolets et régler la précision à 0.2 donne de très bons résultats : à part le sextolet, le rythme obtenu correspond à ce qu'un annotateur humain aurait pu transcrire. Pour le quatrième exemple, là encore, la fonction `true-durations` rajoute un silence, ce qui complique la notation.

Globalement, les résultats d'`omquantify`, si on prend le temps d'ajuster les paramètres, sont un peu meilleurs que la meilleure transcription obtenue par notre algorithme, mais notre algorithme a l'avantage de proposer différentes solutions, parmi lesquelles se trouve le plus souvent une solution convenable.

Un exemple intéressant où notre algorithme obtient de meilleurs résultats qu'`omquantify` est celui de la Figure 4.15. Ici, la suite de durées que l'on souhaite quantifier est la suivante (en ms) : $333, 111, 111, 161, 284$. Les pointillés sont placés toutes les 333 ms. Ce rythme correspond donc au premier rythme de la Figure 4.15 b), où la dernière note a été légèrement retardée. La solution renvoyée par `omquantify` utilise un septolet, car elle minimise l'erreur sur le temps entier. A l'inverse, notre algorithme va fonctionner subdivision par subdivision : dans le deuxième tiers du temps, cela vaut la peine de subdiviser à nouveau, mais dans le troisième tiers, ce n'est pas la peine, le fait de subdiviser encore ferait trop monter la complexité par rapport au gain de précision que cela représente. On ne subdivise donc pas plus loin et on obtient une notation facile à lire, sans dénaturer le rythme d'entrée.

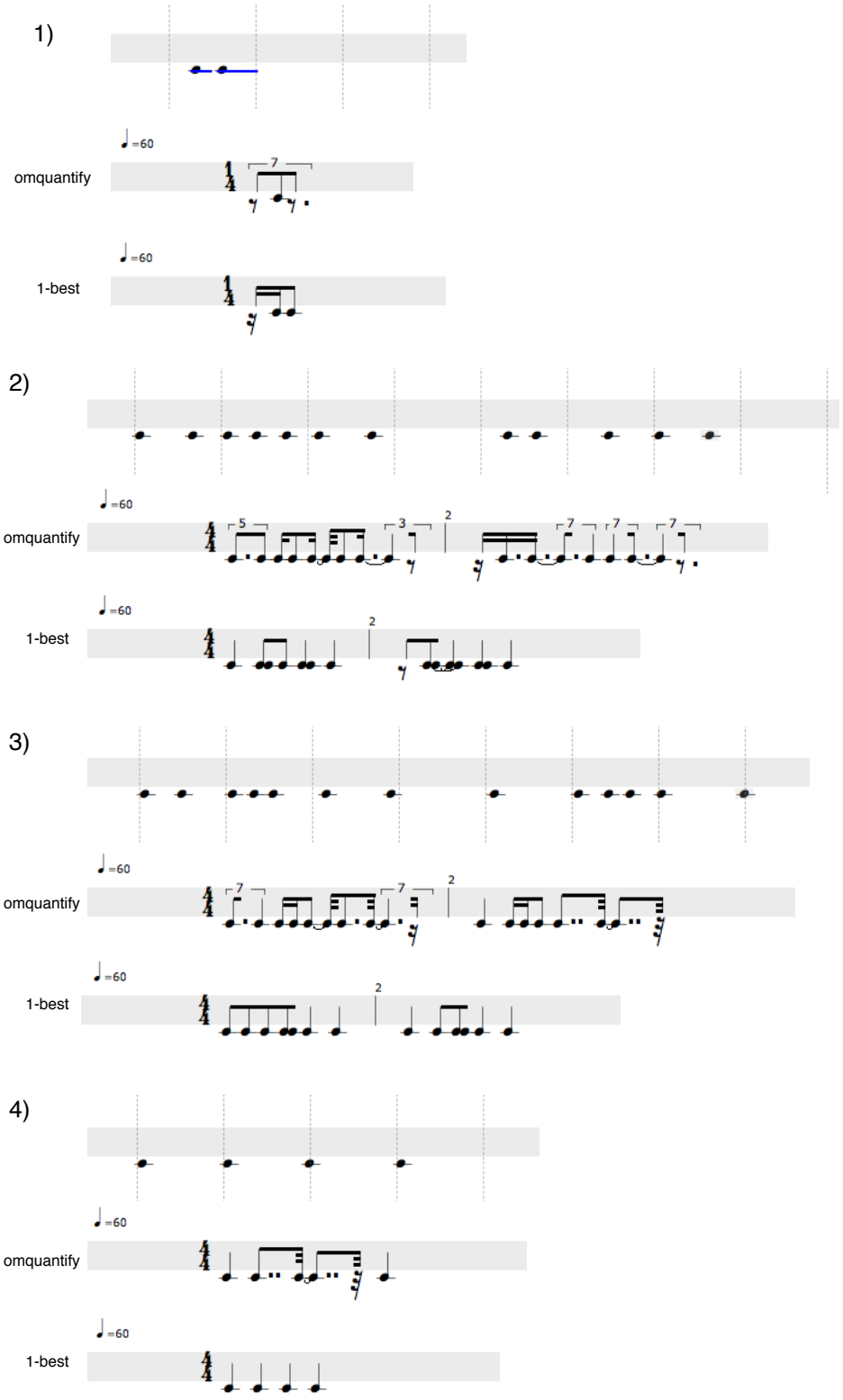

FIGURE 4.13 – Résultats renvoyés par la fonction `omquantify`, comparés aux meilleurs résultats renvoyés par notre algorithme. Les données ont été auparavant mises en forme à l'aide de la fonction `true-durations`.

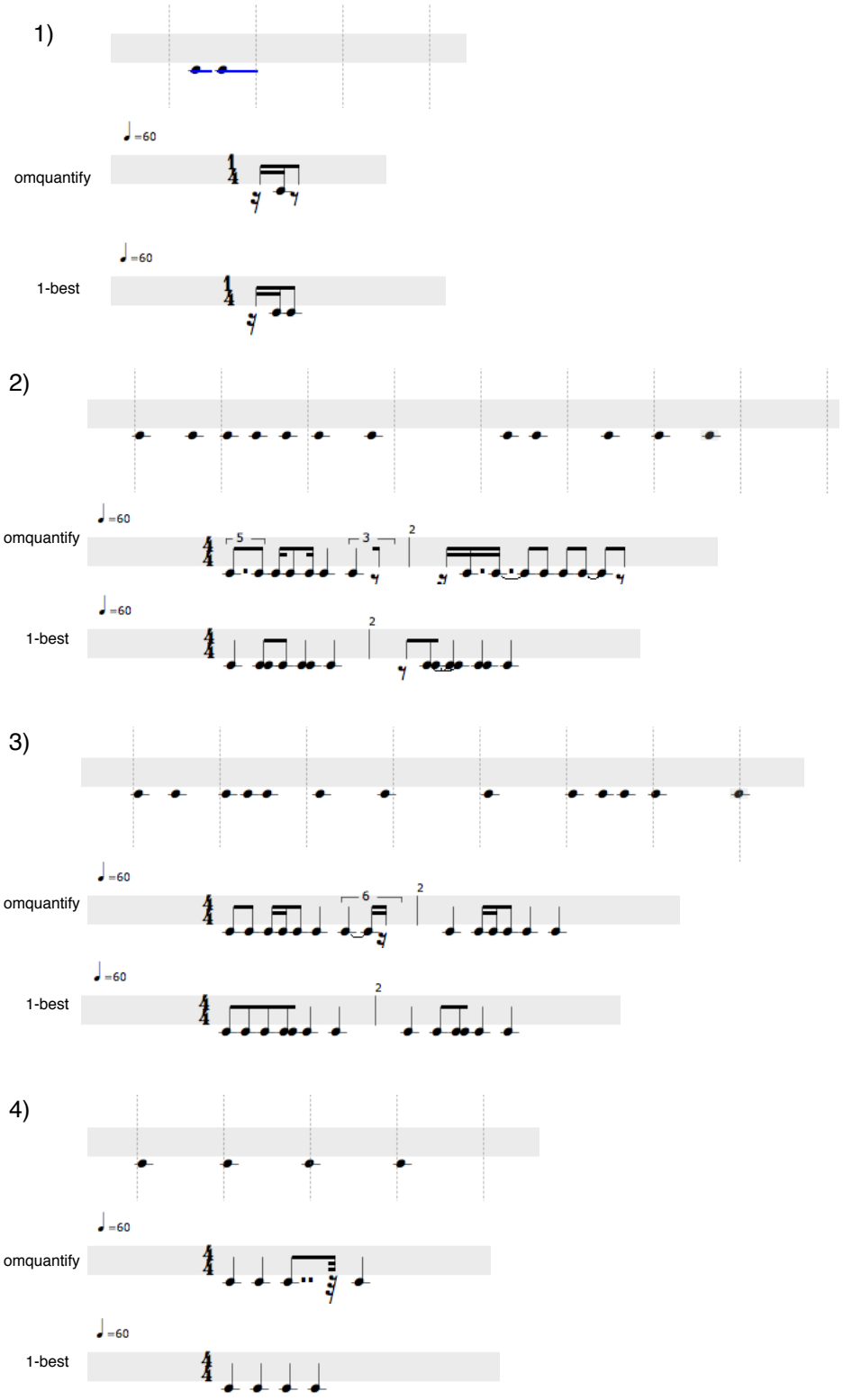

FIGURE 4.14 – Résultats renvoyés par la fonction `omquantify` en ajustant les paramètres, comparés aux meilleurs résultats renvoyés par notre algorithme.

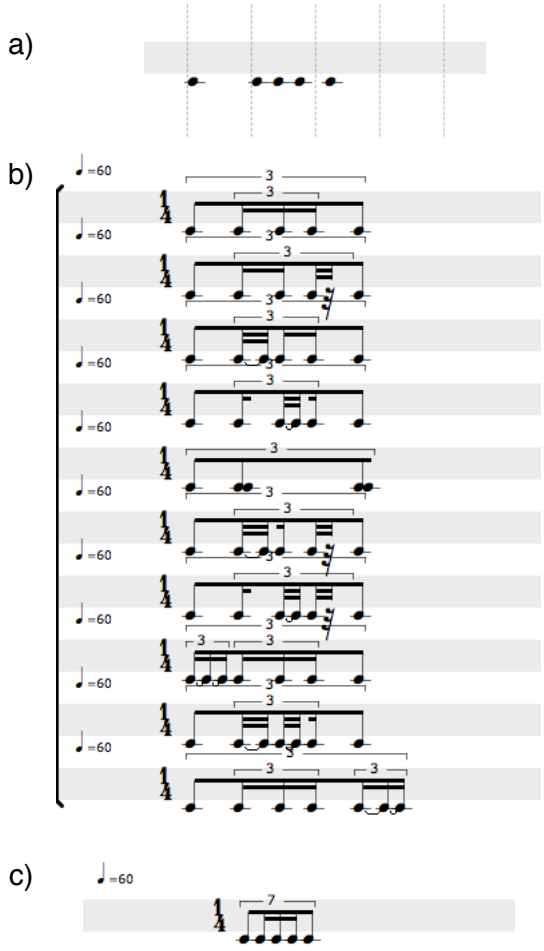

FIGURE 4.15 – a) Entrée à quantifier. b) Résultats renvoyés par notre algorithme. c) Résultat renvoyé par la fonction `omquantify`

Chapitre 5

Bilan et perspectives

5.1 Bilan

Notre stage au sein de l'équipe Représentations Musicales à l'Ircam était consacré à la quantification rythmique dans le contexte particulier de l'environnement de composition assistée par ordinateur OpenMusic.

Dans ce rapport, après une rapide explication sur le rythme, nous avons délimité le problème de la quantification rythmique, décrit quelques formalismes de représentation arborescente du rythme, et présenté des solutions existant pour ce problème dans le chapitre 1.

Nous avons interrogé des utilisateurs d'OpenMusic pour connaître leurs usages des outils de quantification, ainsi que les éventuelles améliorations qu'ils souhaiteraient y voir, et nous avons rassemblé les conclusions de ces entretiens dans le chapitre 2. Nous avons pu dégager plusieurs grands axes d'amélioration préconisés par les utilisateurs rencontrés. Les compositeurs évoquent un besoin de contrôle fin du processus de transcription de l'oeuvre composées, puisque ce processus fait partie intégrante de la création de l'oeuvre. Ils évoquent en particulier des interfaces permettant de spécifier au logiciel de quantification des paramètres signifiants d'un point de vue musical. Nous avons également pu remarquer la difficulté de la gestion des silences, pourtant essentiels à la musique, ainsi que des grace notes.

Suite à ces entretiens, nous avons développé un algorithme, appelé algorithme k-best, permettant de répondre en partie aux besoins des utilisateurs. Cet algorithme a été décrit au chapitre 3. Il se base sur le formalisme d'automates d'arbres pondérés et permet d'énumérer, de façon computationnellement efficace, les k arbres de rythme de plus faible poids selon une certaine mesure, prenant en compte la précision et la complexité de la notation à laquelle ils correspondent (le choix de ces mesures a également été discuté).

Dans le chapitre 4, nous avons présenté les différents choix qui ont été faits lors de l'implémentation de l'algorithme k-best, nous avons également déterminé la complexité en temps de notre algorithme et réalisé des essais pour évaluer son temps d'exécution. Cet algorithme est plus long à exécuter que l'algorithme actuel d'OpenMusic, mais offre des perspectives d'usage intéressantes. Nous avons présenté des résultats donnés par cet algorithme et les avons comparés à ceux de l'outil actuel. Les résultats sont comparables, mais la meilleure gestion des silences et des grace note par notre algorithme fait qu'on peut souvent obtenir de meilleurs résultats parmi ses premières propositions qu'en ajustant les paramètres de l'algorithme actuel.

Nous allons dans la fin de ce rapport proposer différentes pistes d'améliorations, à la fois pour rendre notre système plus performant dans ses résultats et pour aborder des problématiques évoquées par les utilisateurs d'OpenMusic qui n'ont pas été étudiées dans ce stage.

5.2 Perspectives

Nous l'avons vu, les résultats sont meilleurs qu'avec les outils existants dans certains cas, en particulier lorsque l'on souhaite quantifier plus ou moins finement à différents endroits d'un segment, mais la rapidité d'exécution est bien meilleure pour les outils existants. Dans cette partie, nous discuterons des différentes améliorations que l'on pourrait envisager pour améliorer les résultats de la quantification.

5.2.1 Critères de choix des arbres

Nous avons vu que le choix des différents critèes qui permettent de classer les arbres sont absolument déterminants. Une étude plus approfondie de ces critères serait donc un premier moyen d'améliorer les résultats de l'algorithme

5.2.1.1 Choix des mesures de distance

La mesure de distance que nous utilisons est une mesure assez simple : c'est la somme des erreurs de quantification. Choisir une autre fonction de distance pourrait être intéressant. En particulier, ici, on n'a aucune information sur la façon dont l'erreur est répartie. Une seule grosse erreur peut donner la même valeur de précision que plusieurs erreurs réparties. Or, perceptivement, dans le cas où il n'y a qu'une seule grosse erreur, on aura l'impression que le rythme de sortie est très différent du rythme d'entrée, alors que dans le cas où l'erreur est répartie, la différence sera moins sensible. Il faudrait donc trouver des mesures de précision qui privilégient les cas où l'erreur est répartie. On pourrait penser en particulier à calculer la similarité cosinus [21] entre le vecteur des instants d'entrée et celui des instants de sortie, qui a pour propriété d'être insensible aux homothéties, ou encore adapter l'idée des rapports entre durées successives explorée dans [19], mais dans les deux cas, il faudra adapter ces mesures, puisque notre algorithme est récursif et n'a jamais de connaissance globale de la séquence à quantifier, sa vision est restreinte au segment relatif au noeud courant. On pourra également envisager de prendre la somme des carrés des erreurs ou de leurs cubes, comme cela a été fait en particulier dans [1], pour désavantager plus fortement les gros écarts par rapport aux petits.

5.2.1.2 Choix des mesures de complexité

Pour la mesure de complexité, on peut également imaginer d'autres mesures de complexité que celles utilisées. En particulier, on pourrait réaliser des analyses sur des corpus de partitions pour déterminer quels motifs, quels arbres ou sous-arbres de profondeur donnée sont les plus fréquents, et estimer que plus un arbre est fréquent, moins il est complexe. Pour cela, on pourrait s'appuyer sur [22], dans lequel des automates k-testables (analogues du n-gram pour les arbres) sont construits pour rechercher des similarités entre des pièces de musique. On pourrait également essayer de déterminer quand des liaisons vont apparaitre dans le résultat final, ou estimer avec plus de précision le nombre de grace notes, mais là encore, cela suppose d'avoir une connaissance plus globale de l'entrée à quantifier, ce qui sera difficile à réaliser dans le cadre de l'algorithme tel qu'il est conçu. Par contre, un moyen envisageable d'empêcher qu'il n'y ait trop de grace notes serait de pénaliser plus fortement les grace notes lorsqu'elles sont à une faible profondeur de l'arbre que lorsqu'elles sont à une grande profondeur. Cela n'est applicable que quand on sait qu'il y aura une grace note. Ainsi, lorsque l'on a deux entrées alignées à la même borne de la subdivision, l'algorithme préfèrera subdiviser encore quand la subdivision correspond à une valeur longue, comme une ronde ou une blanche, et préfèrera noter une grace note quand il s'agit d'une valeur courte, comme une croche ou une double croche.

5.2.1.3 Choix de la fonction de poids

La fonction de poids utilisée, c'est à dire la manière dont sont combinées les mesures de distance et de complexité pour déterminer le poids d'une solution, est elle aussi très simple, on pourrait essayer de considérer des fonctions plus évoluées. Pour cela, il faudrait étudier de près quelles fonctions permettent vérifier la propriété de monotonie, et c'est un problème qui n'est pas évident. Si l'on garde l'idée d'une simple combinaison linéaire, alors il faudrait faire des tests plus conséquents, sur de plus grands corpus, pour déterminer quels coefficients permettent d'obtenir les meilleurs résultats. En particulier, il serait intéressant de réaliser des tests sur le corpus Kostka-Payne [23], qui contient à la fois des données quantifiées et des performances de pièces extraites de [14], pour obtenir, via des techniques d'apprentissage automatique, les coefficients qui minimisent les erreurs sur ce corpus. Rendre ces coefficients paramétrables par l'utilisateur, par exemple en ajoutant un paramètre permettant de régler l'influence relative des mesures de distance et de complexité permettrait un contrôle plus fin du compositeur sur les résultats obtenus.

5.2.2 Sur les automates

5.2.2.1 Stockage en mémoire des automates

Actuellement, à chaque fois qu'un nouvel automate est construit, on détruit le précédent. Or il arrive souvent que l'on construise un nouvel automate : à chaque fois que l'entrée ou le schéma de subdivision sont modifiés, un nouvel automate est construit, et quand on quantifie des entrées qui durent plusieurs mesures, on construit un nouvel automate pour chaque mesure. De plus, la construction d'un nouvel automate est très coûteuse : en plus de devoir reconstruire la table de hachage et initialiser les listes de candidats, on perd toutes les solutions qui ont été calculées auparavant. En effet, si par exemple, on veut renvoyer les 10 meilleures solutions pour une entrée de deux mesures, les 10 meilleures seront calculées pour la première mesure, puis on construit un nouvel automate, et les 10 meilleures pour la deuxième. Si on souhaite ensuite obtenir la $11^{\text{ème}}$ meilleure solution pour la première mesure, il faudra recalculer les 10 premières avant de pouvoir calculer la $11^{\text{ème}}$. On perd donc l'un des principaux avantages que nous fournissait l'akgorithme utilisé : le fait qu'il soit peu coûteux de générer de nouvelles solutions une fois l'automate créé et les premières solutions trouvées. Trouver un système pour stocker les automates en mémoire serait donc une amélioration primordiale du point de vue de la rapidité d'exécution de l'algorithme. On pourrait par exemple stocker un nombre fixe d'automates et supprimer au fur et à mesure ceux qui ont été le moins utilisés, ou bien laisser le soin à l'utilisateur de supprimer les automates dont il ne souhaite plus se servir.

5.2.2.2 Choisir un autre automate

Dans la construction de l'automate, nous créons un état par chemin, et les entrées à l'intérieur de ce segment sont alignées sur la borne la plus proche. Or, nous l'avons vu à la partie 4.4.1, il arrive que dans une subdivision contenant deux entrées x_i et x_{i+1}, correspondant respectivement au début d'une note et au début d'un silence, toutes les deux situées plus proches de la borne de gauche que de celle de droite, le calcul de la distance soit fait en alignant les deux entrées à gauche, mais l'arbre reconstruit à la sortie aligne x_i à gauche et x_{i+1} à droite, ce qui fausse les résultats. Une première solution à ce problème serait de modifier la fonction de complexité pour qu'elle pénalise fortement ces cas, en particulier lorsqu'ils se produisent à la base de l'arbre et donc engendrent des différences importantes, afin de forcer l'algorithme à subdiviser plus profondément. Une autre solution serait de modifier l'automate pour ne plus contraindre les onsets à être alignés à la borne la plus proche, mais laisser l'algorithme décider de la meilleure solution.

Pour cela, il faudrait, pour chaque chemin, créer non plus un seul état, mais plusieurs, en fonction du nombre d'entrées contenue dans le segment. Ainsi, si le segment correspondant à un chemin contient 3 entrées x_1, x_2 et x_3, on créera 4 états : un état qui aligne toutes les entrées à

gauche, un état qui aligne x_1 et x_2 à gauche, et x_3 à droite, un état qui aligne x_1 à gauche, et x_2 et x_3 à droite, et un état qui aligne toutes les entrées à droite. A chaque état, on calcule ensuite la distance $|x_i - y_i|$. La somme des distance point à point sera évidemment la plus petite lorsque chaque point est aligné à la borne la plus proche, mais en adaptant les valeurs de complexité, on pourrait obtenir d'autres résultats intéressants.

Il faut noter que choisir cette nouvelle forme d'automate augmentera grandement le nombre d'états dans l'automate. On autorise plus de solutions, on augmente donc les chances de trouver la bonne, mais il est aussi plus coûteux de les explorer. D'après l'étude de la partie 4.3, la taille de l'automate est critique pour le temps d'exécution de l'algorithme, une telle solution ne sera donc pas forcément recommandable, surtout pour la quantification d'entrées longues. En particulier, le nombre d'états ne dépendra plus uniquement du schéma de subdivision, mais augmentera aussi avec le nombre d'entrées à aligner, ce qui n'était pas vraiment le cas jusqu'à présent.

5.2.3 Interfaces

La question des interfaces d'utilisation des logiciels, qui était un verrou souvent évoqué lors des entretiens avec les utilisateurs d'OpenMusic, n'a pas pu être abordée en profondeur au cours de ce stage. Néanmoins, nous pouvons proposer des idées pour concevoir un outil plus facilement manipulable par les utilisateurs.

5.2.3.1 Préciser les paramètres significatifs

Un des besoins évoqués par les utilisateurs est celui de pouvoir indiquer à l'algorithme, plutôt que la signature et le tempo, qui ne sont pas forcément connus à priori, des paramètres porteurs de sens musical. En particulier, fixer les barres de mesure, ou les notes sur lesquelles sont censés tomber les temps forts de la musique aurait plus de sens, puisque ce sont des paramètres dont les compositeurs ont en général une idée avnt la quantification.

On pourrait par exemple proposer un outil qui permettrait de placer certains temps forts, à la manière du pré-traitement qui est réalisé par AR et décrit par la Figure 2.2. Une interface dans OpenMusic permettant de segmenter un flux de notes a été proposée dans [5], elle est en particulier utilisée par OM (cf partie 2.2) dans ses analyses pour segmenter en mesures. Cet interface pourrait être enrichie de diverses fonctions spécifiques à la quantification rythmique. L'utilisateur placerait par exemple quelques temps forts via cette interface affichant la série de notes, puis un algorithme proposerait des valeurs de tempo, et afficherait les pulsations correspondantes sur la série de notes. L'utilisateur peut ainsi facilement visualiser la valeur de tempo qui lui semble la meilleure. Il peut ensuite déplacer certaines pulsations pour les faire coïncider avec des débuts ou des fins de notes. Une fois ces ajustements réalisés, l'outil retourne une série d'instants de début et une série de durées recalées, ainsi que la valeur de tempo correpondant, qui peuvent ensuite être utilisés par l'algorithme de quantification. Cette étape permettrait également d'éliminer certains problèmes liés au découpage en mesures évoqué à la partie 4.1.2, en particulier lorsque la première note d'une mesure est jouée légèrement en avance et donc un petite partie déborde sur la mesure précédente.

Sur cette même interface, on pourrait également placer les barres de mesure, et ainsi, en déduire les signatures. On peut avoir deux approches. Soit on considère que l'utilisateur donne toutes les barres de mesures. Dans ce cas, en déduire les signatures est facile, il suffit de compter les temps entre chaque barre de mesure. On peut aussi considérer qu'il n'en donne que quelques unes. Retrouver toutes les barres de mesure est alors plus compliqué, mais peut être fait en trouvant des diviseurs communs des nombres de temps séparant deux barres de mesures.

On obtiendrait alors facilement tous les paramètres nécessaires à la quantification, en les fixant à l'aide d'une interface plutôt que simplement en essayant plusieurs jeux de paramètres.

5.2.3.2 Editeur de schémas de subdivision

Le format des schémas de subdivision est relativement compliqué. Il est difficile à la fois à écrire et à relire, et est très sujet à erreur. De plus, il n'y a actuellement pas de moyen de vérifier que le schéma donne bien l'automate que l'on souhaite, à moins d'afficher toutes les entrées de la table de hachage. Développer une interface facilitant l'écriture et la lecture des schémas serait d'une grande aide pour les utilisateurs. On pourrait par exemple représenter le schéma par un graphe orienté, où des flèches indiquent pour chaque subdivision quelles sont les prochaines subdivisions possibles (en prenant garde de ne pas créer de cycles, auquel cas l'automate serait de taille infinie). Un exemple est fourni en Figure 5.1. On pourrait aussi proposer un certain nombre de schémas de subdivision classiques, que l'utilisateur combinerait ensuite selon ce qu'il souhaite obtenir.

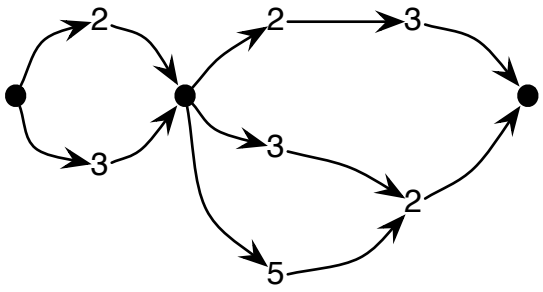

FIGURE 5.1 – Représentation du schéma de subdivision ((2|3) ((2 3) | ((3|5) 2))) par un graphe orienté.

5.2.4 Apprentissage des préférences de l'utilisateur

Une piste qui avait été explorée dans [17] était d'apprendre les préférences de l'utilisateur. Le postulat de départ est que le fait de transcrire une idée musicale en partition fait partie du processus de création d'une oeuvre, et donc chaque compositeur a un style de notation qui lui est propre. A partir des choix faits par l'utilisateur parmi les différentes quantifications proposées par l'outil que nous avons développé, on peut apprendre au fur et à mesure le style de notation de l'utilisateur. Par exemple, à l'aide d'une interface, l'utilisateur choisit parmi les propositions renvoyées par l'algorithme celle qu'il souhaite garder, et celles au contraire qu'il ne souhaite plus voir apparaitre. Les solutions éliminées ne devront plus être proposées par l'algorithme, et celles qui sont sélectionnées devront avoir un poids plus faible que celles qui n'ont pas été sélectionnées. L'utilisateur peut aussi décider de ne rejeter qu'une partie de la solution : si dans une mesure à deux temps, on a une noire et un quintolet de croches, l'utilisateur peut préciser que c'est le quintolet qu'il refuse de voir apparaître à nouveau. Ensuite, en analysant les caractéristiques des solutions retenues ou rejetées, telles que la profondeur de l'arbre, le nombre de grace notes, le nombre de liaisons, les nombres d'occurences de chaque arité, on en déduit les caractéristiques des solutions retenues et éliminées.

On peut ensuite utiliser ces préférences dans l'algorithme, en utilisant des techniques d'apprentissage de langage d'arbres pondérés, comme une nouvelle mesure de complexité des solutions pour améliorer la génération de solutions. On a alors un automate pour la mesure de précision, construit à chaque mesure, qui ne dépend que de l'entrée et du schéma de subdivision, et un autre automate pour la mesure de complexité, qui est appris au fil des choix de l'utilisateur.

Une autre possibilité est d'utiliser cette connaissance non pas dans l'algorithme, mais plutôt comme un post-traitement, permettant d'éliminer d'office certaines solutions, et générer les suivantes sans que l'utilisateur n'intervienne. Cette deuxième option permet de prendre en compte

les liaisons et les grace notes, deux paramètres qui, nous l'avons vu, sont difficilement évaluables pendant le déroulement de l'algorithme mais qui le sont facilement une fois l'arbre reconstruit.

Une dernière possibilité serait d'utiliser ces choix pour apprendre un schéma de subdivision. Il s'agit ici d'un problème difficile, mais qui serait très bénéfique, car comme noue l'avons vu, le format du schéma de subdivision est difficile à manipuler. De plus, le schéma est utilisé dans la construction des automates, donc modifier le schéma implique de reconstruire tous les automates, et il est plus difficile d'en évaluer plusieurs.

Bibliographie

[1] Carlos Agon, Gérard Assayag, Joshua Fineberg, and Camilo Rueda. Kant : A critique of pure quantification. In *Proceedings of the International Computer Music Conference*, pages 52–9, 1994.

[2] Carlos Agon, Karim Haddad, and Gérard Assayag. Representation and rendering of rhythm structures. In *Web Delivering of Music, 2002. WEDELMUSIC 2002. Proceedings. Second International Conference on*, pages 109–113. IEEE, 2002.

[3] Gérard Assayag, Camilo Rueda, Mikael Laurson, Carlos Agon, and Olivier Delerue. Computer-assisted composition at IRCAM : From PatchWork to OpenMusic. *Computer Music Journal*, 23(3) :59–72, 1999.

[4] Marc Bezem, Jan Willem Klop, and Roel de Vrijer. *Term rewriting systems*. Cambridge University Press, 2003.

[5] Jean Bresson and Carlos Pérez-Sancho. New Framework for Score Segmentation and Analysis in OpenMusic. In *Sound and Music Computing*, pages 1–1, Copenhagen, Denmark, 2012. cote interne IRCAM : Bresson12c.

[6] Ali Taylan Cemgil, Peter Desain, and Bert Kappen. Rhythm quantization for transcription. *Computer Music Journal*, 24(2) :60–76, 2000.

[7] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on : `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[8] Peter Desain and Henkjan Honing. The quantization of musical time : A connectionist approach. *Computer Music Journal*, pages 56–66, 1989.

[9] Pierre Donat-Bouillud. Transcription rythmique dans OpenMusic. Stage, 2013.

[10] Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of weighted automata*. Springer Science & Business Media, 2009.

[11] John E Hopcroft and Jeffrey D Ullman. Formal languages and their relation to automata. 1969.

[12] Liang Huang and David Chiang. Better k-best parsing. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 53–64. Association for Computational Linguistics, 2005.

[13] Florent Jacquemard, Pierre Donat-Bouillud, and Jean Bresson. A structural theory of rhythm notation based on tree representations and term rewriting. In *Mathematics and Computation in Music*, volume 9110, page 12. Springer, 2015.

[14] Stefan Kostka and D Payne. Workbook for tonal harmony. *New York*, 1995.

[15] Mikael Laurson. *PATCHWORK : A visual programming language and some musical applications*, volume 6. Sibelius Academy Helsinki, 1996.

[16] Christopher S Lee. The rhythmic interpretation of simple musical sequences : towards a perceptual model. *Musical structure and cognition*, 3 :53–69, 1985.

[17] Adrien Maire. Quantification musicale avec apprentissage sur des exemples. 2013.

[18] Benoit Meudic. *Détermination automatique de la pulsation, de la métrique et des motifs musicaux dans des interprétations à tempo variable d'oeuvres polyphoniques*. PhD thesis, Paris 6, 2004.

[19] Declan Murphy. Quantization revisited : a mathematical and computational model. *Journal of Mathematics and Music*, 5(1) :21–34, 2011.

[20] Geoffroy Peeters. Template-based estimation of time-varying tempo. *EURASIP Journal on Applied Signal Processing*, 2007(1) :158–158, 2007.

[21] Ali Mustafa Qamar. *Generalized Cosine and Similarity Metrics : A Supervised Learning Approach based on Nearest Neighbors*. Theses, Université de Grenoble, 2010.

[22] David Riso-Valero. *Symbolic Music Comparison with Tree Data Structure*. PhD thesis, Ph. D. Thesis, Universitat d'Alacant, Departamento de Lenguajes y Sistemas Informaticos, 2010.

[23] David Temperley. An evaluation system for metrical models. *Computer Music Journal*, 28(3) :28–44, 2004.

[24] David Zicarelli and M Puckett. Max/msp. *Cycling*, 74 :1990–2001, 2002.